14

Класове

Класовете са нови типове данни, дефинирани от потребителя. Те могат да обогатяват възможностите на вече съществуващ тип или да представят напълно нов тип данни.

Класовете са подобни на структурите, даже може да се каже, че в някои отношения са почти идентични. В C++ класът може да се разглежда като структура, на която са наложени някои ограничения по отношение на правата на достъп. Отначало ще разгледаме основното, което е приложимо както за класовете, така и за структурите. Затова ще останем в познатите означения на структурите.

14.1 Пример за програма, която дефинира и използва клас

Основен принцип на процедурното програмиране е модулния. Програмата се разделя на "подходящи" взаимносвързани части (функции, модули), всяка от които се реализира чрез определени средства. Важен е обаче начинът, по който да става определянето на частите и връзките помежду им. Целта е, следващи промени в представянето на данните да не променят голям брой от модулите на програмата. Разсъждения в тази посока довеждат до подхода *абстракция със структури от данни*, който вече разгледахме. Ще напомним, че при него методите за използване на данните се разделят от методите за тяхното конкретно представяне. Програмите се конструират така, че да работят с "абстрактни данни" – данни с неуточнено представяне. След това представянето се конкретизира с помощта на множество функции, наречени **конструктори, мутатори** и **функции за достъп**, които реализират "абстрактните данни" по конкретен начин. Така при решаването на даден проблем се оформят следните нива на абстракцията:

Добра реализация на подхода е тази, при която всяко ниво използва единствено средствата на предходното. Предимствата са, че възникнали промени на едно ниво ще се отразят само на следващото над него. Например, промяна на представянето на данните ще доведе до промени единствено на реализацията на някои от конструкторите, мутаторите или функциите за достъп.

Да се върнем към задачата за рационално-цифрова аритметика. Като използваме подхода абстракция със структури от данни искаме да дефинираме тип данни “рационално число”, след което да го използваме за събиране, изваждане, умножение и деление на рационални числа.

След анализ на правилата, реализиращи тези операции, в глава 11 стигнахме до необходимостта от реализирането на следните примитивни операции за работа с рационални числа:

- конструиране на рационално число по зададени две цели числа, представящи съответно неговите числител и знаменател;
- извличане на числителя на дадено рационално число;
- извличане на знаменателя на дадено рационално число.

Към тях ще добавим и функциите:

- промяна на стойността на рационално число чрез въвеждане, например;
- извеждане на рационално число.

Реализирането на подхода абстракция със структури от данни в този случай показва следните четири нива на абстракция

Ще започнем с реализирането на нивата отдолу нагоре.

Избор на представяне на рационално число

Тъй като рационалното число е частно на две цели числа, можем да го определим чрез структурата:

```
struct rat
{int numer;
 int denom;
};
```

където полето numer означава числителя, а полето denom – знаменателя на рационално число. Тези две полета се наричат **член-данни**, само **данни** или още **абстрактни данни** на структурата. Те

определят множеството от стойности на типа rat, който дефинираме. Трябва да добавим и някакви операции и вградени функции, които да могат да се изпълняват над данни от тип rat. Това ще постигнем с реализацията на следващите две нива на абстракция, определени по-горе.

Реализиране на примитивните операции

Като компоненти на структурата rat ще добавим набор от примитивни операции: конструктори, мутатори и функции за достъп. Ще ги реализираме като член-функции.

а) конструктори

Конструкторите са член-функции, чрез които се инициализират променливите на структурата. Имената им съвпадат с името на структурата. Ще дефинираме два конструктора на структурата rat:

rat() – конструктор без параметри и

rat(int, int) – конструктор с два цели параметъра.

Първият конструктор се нарича още **конструктор по подразбиране**. Използва се за инициализиране на променлива от тип rat, когато при дефиницията ѝ не са зададени параметри. Ще го дефинираме така:

```
rat::rat()
{numer = 0;
 denom = 1;
}
```

Пример: След дефиницията

```
rat p = rat();
```

или съкратено

```
rat p;
```

p се инициализира с рационалното число 0/1.

Вторият конструктор

```
rat::rat(int x, int y)
{numer = x;
 denom = y;
}
```

позволява променлива величина от тип rat да се инициализира с указана от потребителя стойност.

Примери: След дефиницията

```
rat p = rat(1,3);
```

p се инициализира с 1/3, а дефиницията

```
rat q(2,5);
```

инициализира q с 2/5.

Ще отбележим, че и двата конструктора имат едно и също име, но се различават по броя на параметрите си. В този случай се казва, че функцията rat е **предефинирана**.

Декларацията на структура може да съдържа, но може и да не съдържа конструктори.

б) мутатори

Това са функции, които променят данните на структурата. Ще дефинираме мутатора read(), който въвежда от клавиатурата две цели числа (второто различно от нула) и ги свързва с абстрактните данни numer и denom.

```
void rat::read()
{cout << “numer: “;
 cin >> numer;
 do
 {cout << “denom: “;
  cin >> denom;
 } while (denom == 0);
}
```

След обръщението

```
p.read();
```

стойността на p *се променя* като полетата й numer и denom се свързват с въведените от потребителя стойности за числител и знаменател съответно.

в) функции за достъп

Тези функции **не променят** член-данните на структурата, а само извличат информация за тях. Последното е указано чрез използването на запазената дума const, записана след затварящата скоба на формалните параметри и пред знака ;. Ще дефинираме следните функции за достъп:

```
int get_numer() const;
int get_denom() const;
void print() const;
```

Първата от тях извлича числителя, втората – знаменателя, а третата извежда върху екрана рационалното число numer/denom. Реализациите им имат вида:

```
int rat::get_numer() const
{return numer;
}
int rat::get_denom() const
{return denom;
}
void rat::print() const
{cout << numer << “/” << denom << endl;
}
```

След включване на *прототипите* на тези конструктори, мутатори и функции за достъп във фигурните скоби на декларацията на структурата rat, получаваме:

```
struct rat
{int numer;
 int denom;
 // конструктори
 rat();
 rat(int, int);
 // мутатор
 void read();
 // функции за достъп
 int get_numer() const;
 int get_denom() const;
 void print() const;
};
```

След направените дефиниции са възможни следните действия над рационални числа:

```
// p се инициализира с 0/1, q – с 1/6, а r – с 5/9
   rat p, q(1,6), r=rat(5,9);
// p се извежда чрез данновите полета на структурата rat
   cout << p.numer << "/"
        << p.denom << endl;
// q се извежда като се използват
// функциите за достъп до компонентите му
   cout << q.get_numer() << "/"
```

```
        << q.get_denom() << endl;
   // q се извежда чрез функцията за достъп print()
      q.print();
   // p се модифицира чрез мутатора read()
      p.read();
```

С това завършихме реализирането на двете най-долни нива на абстракция и преминаваме към следващото ниво.

Реализиране на правилата за рационално-цифрова аритметика

Като използваме дефинираните конструктори, мутатори и функции за достъп, ще реализираме функциите:

```
   rat sum(rat const &, rat const &);
   rat sub(rat const &, rat const &);
   rat prod(rat const &, rat const &);
   rat quot(rat const &, rat const &);
```

извършващи рационално-числовата аритметика. Функцията sum може да се дефинира по следния начин:

```
   rat sum(rat const& r1, rat const& r2)
   {rat r(r1.get_numer()*r2.get_denom() +
          r2.get_numer()*r1.get_denom(),
          r1.get_denom()*r2.get_denom());
    return r;
   }
```

Другите функции се реализират по аналогичен начин.

Ще отбележим, че по подразбиране, членовете на структурата (член-данни и член-функции) са видими навсякъде в областта на структурата. Това позволява член-данните да бъдат използвани както от примитивните конструктори, мутатори и функции за достъп така и от функциите, реализиращи рационално-числова аритметика.

Например, функцията sum, дефинирана по-горе може да се реализира и така:

```
   rat sum(rat const& r1, rat const& r2)
   {rat r(r1.numer*r2.denom + r2.numer*r1.denom,
          r1.denom*r2.denom);
    return r;
   }
```

Нещо повече, освен чрез мутаторите, член-данните могат да бъдат модифицирани и от външни функции.

Последното противоречи на идеите на подхода абстракция със структури от данни, в основата на който лежи независимостта на използването от представянето на структурата от данни. Това води до идеята да се забрани на модулите от трето и четвърто ниво пряко да използват средствата от първо ниво на абстракция.

Езикът C++ позволява да се ограничи свободата на достъп до членовете на структурата като се поставят подходящи спецификатори на достъп в декларацията ѝ. Такива спецификатори са private и public. Записват се като етикети. Всички членове, следващи спецификатора на достъп private, са достъпни само за член-функциите в декларацията на структурата. Всички членове, следващи спецификатора на достъп public, са достъпни за всяка функция, която е в областта на структурата. Ако са пропуснати спецификаторите на достъп, всички членове са public (както е в случая). Има още един спецификатор на достъп – protected, който е еднакъв със спецификатора private, освен ако структурата не е част от йерархия на класовете, което ще разгледаме по-късно.

С цел реализиране на идеите на подхода абстракция със структури от данни, ще променим дефинираната по-горе структура по следния начин:

```
struct rat
{private:
 int numer;
 int denom;
 public:
 rat();
 rat(int, int);
 void read();
 int get_numer() const;
 int get_denom() const;
 void print() const;
};
```

По такъв начин позволяваме член-данните numer и denom да се използват единствено от член-функциите на структурата rat. Операторът

```
cout << p.numer << "/"
```

```
        << p.denom << endl;
```
вече е недопустим.

Следва програмата, която решава задачата.

```
#include <iostream.h>
struct rat
{private:
 int numer;
 int denom;
 public:
 rat();
 rat(int, int);
 void read();
 int get_numer() const;
 int get_denom() const;
 void print() const;
};
rat::rat()
{numer = 0;
 denom = 1;
}
rat::rat(int x, int y)
{numer = x;
 denom = y;
}
void rat::read()
{cout << "numer: ";
 cin >> numer;
 do
 {cout << "denom: ";
  cin >> denom;
 }while(denom==0);
}
int rat::get_numer() const
{return numer;
}
int rat::get_denom() const
{return denom;
}
```

```
void rat::print() const
{cout << numer << "/" << denom << endl;
}
rat sum(rat const &, rat const &);
rat sub(rat const &, rat const &);
rat prod(rat const &, rat const &);
rat quot(rat const &, rat const &);
int main()
{rat p(1,4), q(1,2);
 p.print();
 q.print();
 cout << "sum:\n";
 sum(p,q).print();
 cout << "subtraction:\n";
 sub(p,q).print();
 cout << "product:\n";
 prod(p,q).print();
 cout << "quotient:\n";
 quot(p,q).print();
 return 0;
}
rat sum(rat const& r1, rat const& r2)
{rat r(r1.get_numer()*r2.get_denom()+
       r2.get_numer()*r1.get_denom(),
       r1.get_denom()*r2.get_denom());
 return r;
}
rat sub(rat const & r1, rat const & r2)
{rat r(r1.get_numer()*r2.get_denom()-
       r2.get_numer()*r1.get_denom(),
       r1.get_denom()*r2.get_denom());
 return r;
}
rat prod(rat const & r1, rat const & r2)
{rat r(r1.get_numer()*r2.get_numer(),
       r1.get_denom()*r2.get_denom());
 return r;
}
```

```
rat quot(rat const & r1, rat const & r2)
{rat r(r1.get_numer()*r2.get_denom(),
       r1.get_denom()*r2.get_numer());
 return r;
}
```

Като се използват функциите за рационално-числова аритметика, могат да се реализират различни приложения. Забелязваме обаче, че тази реализация не съкращава рационални числа. За преодоляването на този недостатък е достатъчно да променим конструктора с параметри. За целта реализираме разделяне на числителя x и знаменателя y на най-големия общ делител на abs(x) и abs(y). Новият конструктор има вида:

```
rat::rat(int x, int y)
{if (x == 0 || y==0)
 {numer = 0;
 denom = 1;
 }
 else
 {int g = gcd(abs(x), abs(y));
  if (x>0 && y>0 || x<0 && y<0)
  {numer = abs(x)/g;
   denom = abs(y)/g;
  }
  else
  {numer = -abs(x)/g;
   denom = abs(y)/g;}
  }
}
```

където int gcd(int x, int y) е известната вече функция за намиране на най-големия общ делител на две естествени числа.

Ще отбележим, че ако в горната програма заменим запазената дума struct с class, програмата няма да промени поведението си. Така дефинирахме класа rat:

```
class rat
{private:
    int numer;
    int denom;
 public:
```

```
        rat();
        rat(int, int);
        void read();
        int get_numer() const;
        int get_denom() const;
        void print() const;
    };
```

а дефиницията

```
    rat p, q=rat(1,7), r(-2,9);
```

определя три негови **обекта**: p, инициализиран с рационалното число 0/1; q, инициализиран с 1/7 и r, инициализиран с -2/9.

Спецификаторът private, забранява използването на член-данните numer и denom извън класа. Получава се *скриване* на информация, което се нарича още **капсолиране на информация**. Член-функциите на класа rat са обявени като public. Те са видими извън класа и могат да се използват от външни функции. Затова public-частта се нарича още **интерфейсна част на класа** или само **интерфейс**. Чрез нея класът комуникира с външната среда. Освен функции, интерфейсът може да съдържа и член-данни, но засега ще се стараем това да не се случва.

*Ще отбележим, че конструкторите се използват само когато се създават обекти. Опитите за **промяна** на обект чрез обръщение към конструктор предизвикват грешки.*

Пример:

```
rat q(1,7);    // коректно
q.rat();       // предизвиква грешка
q(2,9);        // предизвиква грешка
q.rat(3,4);    // предизвиква грешка.
```

Забележете, езикът C++ дефинира структурите и класовете почти идентично. Съществената разлика е свързана със спецификаторите на достъп. По подразбиране членовете на структура имат public (публичен) достъп, а членовете на клас – private (частен) достъп. Все пак възниква въпросът: *Защо да има две различни конструкции struct и class, когато разликите са толкова малки?* Причината е свързана с мобилността на програмите, с цел да се запази съвместимостта между езиците C и C++. Бярне Страуструп обяснява, че причината е по-скоро културна, отколкото техническа. Той препоръчва структурите да се използват само когато се реализират

свойства, които са прости и включват малки “натоварвания”. По-точно, когато реализираната структура от данни е идентична с интерфейса си. В останалите случаи да се използват класове.

14.2 Дефиниране на класове

Класовете осигуряват механизми за създаване на напълно нови типове данни, които могат да бъдат интегрирани в езика, а също за обогатяване възможностите на вече съществуващи типове. Дефинирането на един клас се състои от две части:

- декларация на класа и
- дефиниция на неговите член-функции (методи).

14.2.1. Декларация на клас

Декларацията на клас се състои от заглавие и тяло. Заглавието започва със запазената дума class, следвано от името на класа. Тялото е заградено във фигурни скоби. След скобите стои знакът “;” или списък от обекти. В тялото на класа са декларирани членовете на класа (член-данни и член-функции) със съответните им нива на достъп. Фиг. 14.1 илюстрира *непълно* (но достатъчно за целите на настящите разглеждания) синтаксиса на декларацията на клас.

Декларация на клас

```
<декларация_на_клас> ::= <заглавие> <тяло>
<заглавие> ::= class [<име_на_клас>]опц
<тяло> ::= {<декларация_на_член>;
            {<декларация_на_член>;}опц
           }[<списък_от_обекти>]опц;
<декларация_на_член> ::=
     <декларация_на_конструктор>|<декларация_на_мутатор>|
    <декларация_на_функция_за_достъп>|<декларация_на_данна>
<декларация_на_конструктор> ::=
     [<спецификатор_на_достъп>:]опц<име_на_клас>(<параметри>)
<декларация_на_мутатор> ::=
     [<спецификатор_на_достъп>:]опц <тип>
                 <име_на_мутатор>(<параметри>)
<декларация_на_функция_за_достъп> ::=
```

```
       [<спецификатор_на_достъп>:]опц <тип>
                  <име_на_функция_за_достъп>(<параметри>) const;
   <спецификатор_на_достъп> ::= private | public | protected
   <параметри> :: <празно> | void |
                  <параметър> {, <параметър>}опц
   <параметър> ::= <тип> [ &|опц * [const]опц] опц
   <декларация_на_данна>      ::=      <тип>      <име_на_данна>{,
<име_на_данна>}опц
   <тип> ::= <име_на_тип>|<дефиниция_на_тип>
   <списък_от_обекти> ::=
       <обект> [= <име_на_клас>(<фактически_параметри>)]опц
               {,<обект>[=<име_на_клас>(<фактически_параметри>)]оп
ц }опц
               {, <обект>(<фактически_параметри>)}опц
               {, <обект> = <вече_дефиниран_обект>}опц
където <име_на_клас>, <име_на_мутатор>, <име_на_данна>, <обект> и
<име_на_функция_за_достъп> са идентификатори, а <фактически_
параметри> е определено в Глава 8.
```

Фиг. 14.1 Декларация на клас

За имената на класовете важат същите правила, които се прилагат за имената на всички останали типове и променливи. Също като при структурите името на класа може да бъде пропуснато. Ще напомним, че използването на долния индекс “опц” означава, че означението е от езика на Бекус-Наур за описание на синтаксиса на език за програмиране.

Имената на членовете на класа са локални за него, т.е. в различни класове в рамките на една програма могат да се дефинират членове с еднакви имена. Член-данни от един и същ тип могат да се изредят, разделени със запетая и предшествани от типа им.

Пример:

```
class point
{private:
  double x, y;    // x и y са член-данни на класа prat
 public:
  point(double, double);
```

```
    void read();       // мутатор със същото име като на класа rat
    int get_x() const;
    int get_y() const;
    void print() const;   // със същото име като на класа rat
  }p=point(2,7), q(-2,3), r=q;
```

Препоръчва се член-данните да се декларират в нарастващ ред по броя на байтовете, необходим за представянето им в паметта. Така за повечето реализации се получава оптимално изравняване до дума.

Забележка: Типът на член-данна на клас **не може** да съвпада с името на класа, но типът на член-функция на клас **може** да съвпада с името на класа.

В тялото, някои декларации на членове могат да бъдат предшествани от **спецификаторите на достъп** private, public или protected. Областта на един спецификатор на достъп започва от спецификатора и продължава до следващия спецификатор. Подразбиращ се спецификатор за достъп е private. Един и същ спецификатор на достъп може да се използва повече от веднъж в декларация на клас.

Препоръчва се, ако секция public съществува, да бъде първа в декларацията, а секцията private да бъде последна в тялото на класа.

Достъпът до членовете на класовете може да се разгледа на следните две нива:

- По отношение на *член-функциите в класа* е в сила, че те имат достъп до всички членове на класа. При това не е необходимо тези компоненти да се предават като параметри. Този режим на достъп се нарича **режим на пряк достъп**. Поради тази причина функциите rat(), read(), print(), get_numer() и get_denom() са без параметри. Освен това член-функцията print() може да бъде дефинирана и по следния начин:

```
  void rat::print() const
  {cout << get_numer() << "/" <<
        << get_denom() << endl;
  }
```

или

```
  void rat::print() const
```

```
{cout << this->get_numer() << "/" <<
      << this->get_denom() << endl;
}
```

Смисълът на this ще бъде обяснен в т. 14.4.5, а на -> - в т.14.5.

- По отношение на *функциите, които са външни за класа*, режимът на достъп са определя от начина на деклариране на членовете.

Членовете на даден клас, декларирани като private (декларирани след запазената дума private) са видими (достъпни) само в рамките на класа. Външните функции нямат достъп до тях. По подразбиране членовете на класовете са private. Това позволява декларацията на класа rat да запишем и по следния начин:

```
class rat
{int numer;
 int denom;
 public:
 rat();
 rat(int, int);
 void read();
 int get_numer() const;
 int get_denom() const;
 void print() const;
};
```

Чрез използването на членове, обявени като private, се постига скриване на членове за външната за класа среда. Процесът на скриване се нарича още **капсолиране на информацията**.

Членовете на клас, които трябва да бъдат видими извън класа (да бъдат достъпни за функции, които не са методи на дадения клас) трябва да бъдат декларирани като public (декларирани след запазената дума public). Всички методи на класа rat са декларирани като public и следователно могат да се използват навсякъде в програмата за работа с рационални числа.

Освен като private и public, членовете на класовете могат да бъдат декларирани и като protected. Тъй като този спецификатор на достъп има отношение към производните класове и процеса на наследяване, разглеждането му засега ще бъде отложено. Ще отбележим, че ако в класа rat заменим private с protected, поведението на класа няма да се промени.

14.2.2 Дефиниране на методите на клас

След декларирането на клас, трябва да се дефинират неговите методи. Дефинициите са аналогични на дефинициите на функции, но името на метода се предшества от името на класа, на който принадлежи метода, следвано от оператора за принадлежност :: (Нарича се още оператор за област на действие). Такива имена се наричат **пълни**. (Операторът :: е ляво-асоциативен и с един и същ приоритет със (), [] и ->). На Фиг. 14.2 е даден синтаксисът на дефиницията на метод на клас.

Дефиниция на метод на клас

<дефиниция_на_метод_на_клас> ::=

[<тип>]$_{опц}$ <име_на_клас>::<име_на_функция>(<параметри>) [const]$_{опц}$

{<тяло>}

<тяло> ::= <редица_от_оператори_и_дефиниции>

където <име_на_клас> и <име_на_функция> са идентификатори, а <параметри> се определя както в дефиниция на функция.

Фиг. 14.2 Дефиниция на метод на клас

Ще отбележим, че дефиницията на конструктор **не започва** с <тип>, а запазената дума const може да присъства само в дефинициите на функциите за достъп. Добрият стил на програмиране изисква използването на const в дефинициите на функциите за достъп и също в техните декларации. Ако се пренебрегне това изискване, могат да се създадат класове, които да не могат да се използват от други програмисти.

Пример: Нека искаме да използваме класа rat, но програмистът му е забравил или нарочно не е декларирал член-функцията print() като const и rat има вида:

```
class rat
{private:
   ...
 public:
   ...
   void print();
```

```
};
```
Нека декларираме класа prat, използващ класа rat, коректно, т.е. функциите за достъп обявяваме като const.
```
class prat
{private:
 int a;
 rat p;  // използване на класа rat
 ...
 public:
 ...
 void print() const;
};
```
където
```
void prat::print() const
{cout << a << endl;
 p.print(); // тази print() е член-функцията на класа rat
};
```
Компилаторът ще съобщи за грешка в обръщението p.print(), защото p е обект на класа rat, а член-функцията rat::print() не е декларирана като const. Компилаторът предполага, че p.print() може да модифицира p. Но p е член-данна на prat, а prat::print() е const, с което твърдо е обещава да не го модифицира.

Обикновено дефинициите на методите са разположени веднага след деклариране на класа, на който те са членове. Възможно е обаче, дефинициите на методите на един клас да бъдат част от декларациите на този клас, т.е. в декларациите на член-функциите на класа могат да се зададат не само прототипите им, но и техните тела.

Пример: Класът rat може да бъде дефиниран и по следния начин:
```
class rat
{private:
  int numer;
  int denom;
 public:
  rat()
  {numer = 0;
   denom = 1;
  }
  rat(int a, int b)
```

```
     {if (a == 0 || b==0)
      {numer = 0;
       denom = 1;
      }
      else
      {int g = gcd(abs(a), abs(b));
       if (a>0 && b>0 || a<0 && b<0)
       {numer = abs(a)/g;
        denom = abs(b)/g;}
       else
       {numer = - abs(a)/g;
        denom = abs(b)/g;
       }
      }
     }
    void read()
    {cout << "numer: ";
     cin >> numer;
     do
     {cout << "denom: ";
      cin >> denom;
     } while (denom == 0);
    }
    int get_numer() const
    {return numer;
    }
    int get_denom() const
    {return denom;
    }
    void print() const
    {cout << numer << "/" << denom << endl;
    }
   };
```

В този случай обаче член-функциите се третират като **вградени (inline) функции**.

Допълнение (вградени функции) С цел повишаване на бързодействието, езикът C++ поддържа т.нар. вградени функции.

Кодът на тези функции не се съхранява на едно място, а се копира на всяко място в паметта, където има обръщение към тях. Използват се като останалите функции, но при декларирането и дефинирането им заглавието им се предшества от модификатора inline.

Пример:

```
#include <iostream.h>
inline int f(int, int); // декларация на вградената функция f
void main()
{cout << f(1,5) << endl;
}
inline int f(int a, int b) // дефиниция на вградената функция f
{return (a+b)*(a-b);
}
```

Ще добавим, че дефиницията на вградена функция трябва да се намира в същия файл, където се използва, т.е. не е възможна разделна компилация, тъй като компилаторът няма да разполага с кода за вграждане. Използването на вградени функции води до икономия на време, за сметка на паметта. Затова се препоръчва използването им само при “кратки” функции. Ще отбележим също, че модификаторът inline е само заявка към компилатора, която може да бъде, но може и да не бъде изпълнена. Възможно е компилаторът да откаже вграждане, ако реши, че функцията е прекалено голяма или има други причини, възпрепятстващи вграждането. Ограниченията за вграждане зависят от конкретния компилатор.

Често член-функциите се реализират като вградени функции. Това увеличава ефективността на програмата, използваща класа. Декларацията на вградени член-функции може да се осъществи и по следния начин:

```
class rat
{private:
  int numer;
  int denom;
 public:
  rat();
  rat(int, int);
  void read();
```

```
    int get_numer() const;
    int get_denom() const;
    void print() const;
};
inline rat::rat()
{numer = 0;
 denom = 1;
}
inline rat::rat(int x, int y)
...
inline void rat::read()
...
inline int rat::get_numer() const
...
inline int rat::get_denom() const
...
inline void rat::print() const
...
```

Телата на някои от член-функциите са пропуснати, тъй като вече са известни.

Ще отбележим също, че в тялото на дефиницията на член-функция явно не се указва обектът, върху който тя ще се приложи. Този обект участва неявно - чрез член-данните на класа. Заради това се нарича **неявен параметър,** а член-данните – **абстрактни данни.** Връзката между неявния параметър и обект ще бъде показана в т. 3. Параметри, които участват явно в дефиницията на член-функция се наричат **явни**. *Всяка член-функция има точно един неявен параметър и нула или повече явни.*

14.2.3 Област на класовете

За разлика от функциите, класовете могат да се декларират на различни нива в програмата: *глобално* (ниво функция) и *локално* (вътре във функция или в тялото на клас).

Областта на глобално деклариран клас започва от декларацията и продължава до края на програмата. Примерите досега бяха с такива класове.

Ако клас е деклариран във функция, всички негови член-функции трябва да са вградени (inline). В противен случай ще се получат функции, дефинирани във функция, което не е възможно.

Пример:

```
void f(int i, int* p)
{int k;
 class CL
 {public:
   // всички методи са дефинирани в тялото на класа
  ...
  private:
  ...
 };
// тяло на функцията f
 CL x;
 ...
}
```

Областта на клас, дефиниран във функция, е функцията. Обектите на такъв клас са видими само в тялото на функцията.

Възможно е използването на обекти (в широкия смисъл на думата) с еднакви имена. В сила е правилото, че в областта си локалният обект скрива нелокалния.

Не е възможно в тялото на локално дефиниран клас да се използва функцията, в която класът е дефиниран.

Пример: Нека сме в означенията на горния пример.

```
void f(...)
{...
 class cl
 {
   // не може да се използва функцията f
 };
 ...
}
```

14.3 Обекти

След като даден клас е дефиниран, могат да бъдат създавани негови екземпляри, които се наричат **обекти**. Връзката между клас и

обект в езика C++ е подобна на връзката между тип данни и променлива, но за разлика от обикновените променливи, обектите се състоят от множество **компоненти** (член-данни и член-функции). На Фиг. 14.3 е даден синтаксисът на дефиниция на обект на клас.

Дефиниция на обект на клас

<дефиниция-на_обект_на_клас> ::=

<име_на_клас> <обект> [=<име_на_клас>(<фактически_параметри>)]$_{опц}$

{, <обект>[=<име_на_клас>(<фактически_параметри>)]$_{опц}$ }$_{опц}$

{, <обект>(<фактически_параметри>)}$_{опц}$

{, <обект> = <вече_дефиниран_обект>}$_{опц}$;

<обект> ::= <идентификатор>

където <фактически_параметри> е определено в Глава 8.

Фиг. 14.3 Дефиниция на обект на клас

Когато за даден клас явно са дефинирани конструктори, при всяко дефиниране на обект на класа те автоматично се извикват с цел да се инициализира обектът. Ако дефиницията е без явна инициализация (например rat p;), дефинираният обект се инициализира според дефиницията на конструктора по подразбиране, ако такъв е определен, и се съобщава за грешка в противен случай. Ако дефиницията е с явна инициализация, обръщението към конструкторите трябва да бъде коректно.

Пример: Дефиницията

```
rat p, q(2,3), r=rat(3,8);
```

определя три обекта: p, инициализиран с рационалното число 0/1, q, инициализиран с 2/3 и r, инициализиран с 3/8. Тя е добре ределена, тъй като класът rat има конструктор по подразбиране и двуаргументен конструктор. Ако елиминираме конструктора по подразбиране в класа rat, горната дефиниция ще съобщи за грешка заради p. Валидна е обаче дефиницията:

```
rat q(2,3), r=rat(3,8);
```

Ако се откажем и от другия конструктор, последната дефиниция също ще стане невалидна.

Когато за даден клас явно не е дефиниран конструктор, реализацията автоматично генерира **подразбиращ се конструктор**. Този конструктор изпълнява редица действия, като заделяне на памет за обектите, инициализиране на някой системни променливи и др. Дефиницията на обект от този клас трябва да е без явна инициализация.

Пример:

```
#include <iostream.h>
class pom
{private:
  int a;
 public:
  int b;
  void read();
  void print() const;
};
void main()
{pom x;  // инициализация според подразбиращия се
         //конструктор, генериран от компилатора на C++
 x.read();
 x.print();
}
void pom::print()const
{cout << "a= "<< a << " b=" << b << endl;
}
void pom::read()
{cout << "a= ";
 cin >> a;
 cout << "b= ";
 cin >> b;
}
```

В случая обектът x се инициализира с неопределена стойност. Опитите за инициализацията му като структура предизвикват грешки.

Декларацията на клас не заделя памет за него. Памет се заделя едва при дефинирането на обект от класа. Дефиницията

```
rat p, q(2, 3), r = rat(3, 8);
```

заделя за обектите p, q и r по 8 байта ОП (по 4В за всяка от данните им numer и denom).

Достъпът до компонентите на обектите (ако е възможен) се осъществява чрез задаване на името на обекта и името на данната или метода, разделени с точка (Фиг. 14.4). Изключение от това правило правят конструкторите (Фиг. 14.5).

Достъп до компонента на обект

<компонента_на_обект> ::= <обект>.<данна>|
<обект>.<име_на_член_функция>()|
<обект>.<име_на_член_функция>(<параметри>)

<име_на_член_функция> е <идентификатор>, означаващ име на мутатор или име на функция за достъп.

Фиг. 14.4 Достъп до компонента на обект

Пример:

```
rat p(1,2), q;
p.get_numer() // достъп до член-функцията get_numer() за обекта p
q.get_numer() // достъп до член-функцията get_numer() за обекта q
```

Ще отбележим също, че на практика обектите p и q нямат свои копия на метода get_numer(). И двете обръщения се отнасят за един и същ метод, но при първото обръщение се работи с данните за обекта p, а при второто – с данните за обекта q.

При създаването на обекти на един клас кодът на методите на този клас не се копира във всеки обект, а се намира само на едно място в паметта.

Естествено възниква въпросът *по какъв начин методите на един клас "разбират" за кой обект на този клас са били извикани*. Отговорът на този въпрос дава указателят this. Всяка член-функция на клас поддържа допълнителен формален параметър - указател с име this и от тип <име_на_клас>*. За да разберем точно как става това, ще разгледаме как компилаторът на C++ обработва член-функция и обръщение към член-функция на клас. Извършват се следните преобразувания:

а) Всяка член-функция на даден клас се транслира в обикновена функция с уникално име и един допълнителен параметър – указателят this.

Пример: Функцията

```
void rat::print()
{cout << numer << "/" << denom << endl;
}
```

се транслира в

```
void print_rat(rat* this)
{cout << this->numer << "/" << this->denom << endl;
}
```

б) Всяко обръщение към член-функция се транслира в съответствие с преобразуването от а).

Пример: Обръщението

```
p.print();
```

се транслира в

```
print_rat(&p);
```

Указателят this може да се използва явно в кода на съответната член-функция, макар че е глупаво да се напише:

```
void rat::print()
{cout << this->numer << "/" << this->denom << endl;
}
```

Като приложение на указателя this ще реализираме функцията

```
rat sum(rat const &, rat const &);
```

като член-функция на класа rat. За целта ще включим псевдонима й

```
rat sum(rat const &, rat const &);
```

в public-секцията на тялото на rat и ще я дефинираме по следния начин:

```
rat rat::sum(rat const & r1, rat const & r2)
{numer = r1.numer*r2.denom+r2.numer*r1.denom;
 denom = r1.denom*r2.denom;
 return *this;
}
```

Нека

```
rat p=rat(1,4), r(1,2), q=rat(1,4);
```

Фрагментът

```
r.sum(p.sum(p, r), q);
r.print();
```

```
p.print();
```
съобщава 6/8 за стойност на p и 32/32 - за стойност на r. Обръщението p.sum(p, r) намира сумата на рационалните числа p и r и я свързва с обекта p, а r.sum(p.sum(p, r), q) събира полученото рационално число с q и свързва резултата с обекта r.

Обекти от един и същ клас могат да се присвояват един на друг. Присвояването може да е и на ниво инициализация (фиг. 14.3).

Пример: Допустими са дефинициите

```
rat p, q(4,5), r=q;
p = q;
...
r = p;
```

При присвояването се копират всички член-данни на обекта. Така присвояването

```
r = p;
```

е еквивалентно на

```
r.numer = p.numer;
r.denom = p.denom;
```

Подробности относно процеса на присвояване са дадени в следващите части на тази глава.

Някои задачи върху дефиниране на класове

Задача 118. Да се дефинира клас "точка в равнината" с две член-данни - двете декартови координати на точката и подходящи член-функции. Като се използва дефинираният клас да се напише програма, която:

а) въвежда n различни точки от равнината, след което ги транслира с (2, 4) и извежда получените точки;

б) намира разстоянието между всеки две точки (все едно старите или новите);

в) намира точките, разстоянието между които е най-малко (най-голямо);

г) проверява, дали въведените точки, в реда, в който са въведени, образуват изпъкнал многоъгълник.

Програма Zad118.cpp решава задачата.

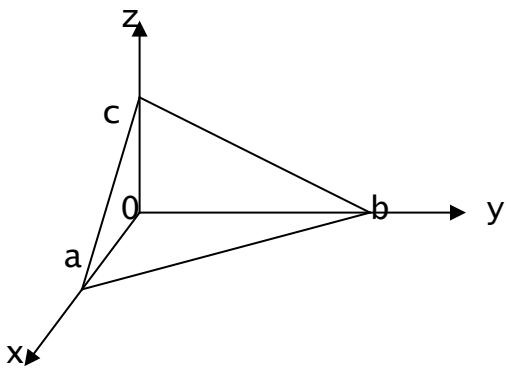
```
// Program Zad118.cpp
#include <iostream.h>
```

```
#include <math.h>
class point
{private:
  double x;
  double y;
 public:
  point(double=0, double=0);
  void read();
  void move(double, double);
  double get_x() const
  {return x;
  }
  double get_y() const
  {return y;
  }
  void print() const;
};
point::point(double a, double b)
{x = a;
 y = b;
}
void point::read()
{cout << "x= ";
 cin >> x;
 cout << "y= ";
 cin >> y;
}
void point::print() const
{cout << "(" << x << ", " << y << ")" << endl;
}
void point::move(double dx, double dy)
{x = x + dx;
 y = y + dy;
}
double dist(point X, point Y)
{return sqrt(pow(Y.get_x()-X.get_x(), 2) +
             pow(Y.get_y()-X.get_y(), 2));
}
```

```
int main()
{// a)
 cout << "n= ";
 int n;
 cin >> n;
 point table[10];
 for (int i=0; i<=n-1; i++)
  table[i].read();
 for (i=0; i<=n-1; i++)
  table[i].print();
 cout << endl;
 for (i=0; i<=n-1; i++)
 {table[i].move(2,4);
  table[i].print();
 } // б)
 for (i=0; i<=n-2; i++)
  for (int j=i+1; j<=n-1; j++)
    cout << dist(table[i], table[j]) << endl;
 return 0;
}
```

Подточки в) и г) оставяме за самостоятелна работа. Разгледайте добре решението и обяснете ролята на конструктора с подразбиращи се параметри. Какво щеше да стане ако параметрите му не бяха подразбиращи се?

Задача 119. Да се дефинират класове “точка в пространството” и “пирамида” с подходящи член-данни и член-функции. Като се използват дефинираните класове да се напише програма, която въвежда n точки в пространството и установява дали всички те принадлежат на пирамидата



включително на контура й.

Програма Zad119. cpp решава задачата.

```
// Program Zad119.cpp
#include <iostream.h>
class Point
{public:
  void Read();
  double GetX() const;
  double GetY() const;
  double GetZ() const;
 private:
  double x, y, z;
};
void Point::Read()
{cout << "x= "; cin >> x;
 cout << "y= "; cin >> y;
 cout << "z= "; cin >> z;
}
double Point::GetX()const
{return x;
}
double Point::GetY()const
{return y;
}
double Point::GetZ()const
{return z;
}
class Piramid
{public:
  void Read();
  bool IsPointIn(Point const &p) const;
 private:
  double a, b, c;
};
void Piramid::Read()
{cout << "a= "; cin >> a;
 cout << "b= "; cin >> b;
 cout << "c= "; cin >> c;
}
```

```
bool Piramid::IsPointIn(Point const &p) const
{double x =  p.GetX()/a + p.GetY()/b + p.GetZ()/c;
 return (p.GetX()>=0 && p.GetY()>=0 && p.GetZ()>=0 &&
         p.GetX()/a + p.GetY()/b + p.GetZ()/c <=1);
}
int main()
{Piramid p;
 cout << "Input piramid \n";
 p.Read();
 Point pt[100];
 cout << "number of points: ";
 int n; cin >> n;
 for (int i=0; i<=n-1; i++)
 {cout << "Point " << i << ": \n";
  pt[i].Read();
 }
 int x = 0;
 while (x<=n-2 && p.IsPointIn(pt[x])) x++;
 if (p.IsPointIn(pt[x])) cout << "Yes\n";
 else cout <<"No\n";
 return 0;
}
```

17.4 Конструктори

Създаването на обекти е свързано с отделяне на памет, запомняне на текущо състояние, задаване на начални стойности и др. дейности, които се наричат **инициализация на обекта**. В езика C++ тези дейности се изпълняват от специален вид член-функции на класовете – конструкторите.

14.4.1. Дефиниране на конструктор

На Фиг. 14.5 е дадена най-често използваната форма за дефиниране на конструктор.

Дефиниране на конструктор (най-често използвана форма)
<дефиниция_на_конструктор> ::=
<име_на_клас>::<име_на_клас>(<параметри>)
{<тяло>}
<тяло> ::= <редица_от_оператори_и_дефиниции>
<параметри> се определя както формални параметри на функция.

Фиг. 14.5 Дефиниране на конструктор (най-често използвана форма)

Ще напомним, че конструкторът е член-функция, която притежава повечето характеристики на другите член-функции, но има и редица особености, като:

- Името на конструктора съвпада с името на класа.
- Типът на резултата е указателят this и явно не се указва.
- Изпълнява се автоматично при създаване на обекти.
- Не може да се извиква явно (обръщение от вида r.rat(1,4) е недопустимо).

Освен това в един клас може явно да не е дефиниран конструктор, но може да са дефинирани и няколко конструктора.

Типична дефиниция на конструктор е дадена в следващия пример.

Пример:

```
class CL
{public:
  CL(int, int, int);
  void print();
  ...
 private:
  int a, b, c;
};
CL::CL(int x, int y, int z) // конструктор с три параметъра
{a = x;
 b = y;
 c = z;
}
```

Класът CL в този пример има един триаргументен конструктор. При създаване на обект на класа, този конструктор ще се изпълнява автоматично, стига да е коректно извикан, в резултат на което член-данните a, b и c на създадения обект ще се свържат със

стойностите, които се подадат като фактически параметри на конструктора.

На Фиг. 14.6 е дадена по-обща дефиниция на конструктор.

Дефиниране на конструктор

<дефиниция_на_конструктор> ::=

<име_на_клас>::<име_на_клас>(<параметри>):

    <член_данна>(<израз>){,<член_данна>(<израз>)}$_{опц}$

{<тяло>}

<тяло> ::= <редица_от_оператори_и_дефиниции>

Фиг. 14.6 Дефиниране на конструктор

Забелязваме, че е възможно член_данна да се свърже с инициализираща стойност в заглавието на конструктора.

Пример: Конструкторът на класа CL може да се дефинира и по следния начин

```
CL::CL(int x, int y, int z): a(x), b(y), c(z)
{}
```

Ще отбележим, че не е задължително всички член-данни да са инициализирани само пред тялото или само вътре в тялото на конструктора.

Пример: Допустима е дефиницията

```
CL::CL(int x, int y, int z): a(x)
{b = y;
 c = z;
}
```

Смисълът на обобщената синтактична конструкция е, че инициализацията на член-данните в заглавието предшества изпълнението на тялото на конструктора. Това я прави изключително полезна. Използването й увеличава ефективността на класа поради следните съображения. Когато член-данни на клас са обекти, в дефиницията на конструктора на класа при инициализацията се използват конструкторите на класовете, от които са обектите (член-данни). Преди да започне изпълнението на указаните конструктори, автоматично се извикват конструкторите по подразбиране на всички член-данни, които са обекти. Веднага след това тези член-данни се инициализират с обектите, резултат от изпълнението на извиканите

конструктори. Това двойно извикване на конструктори намалява ефективността на програмата.

Пример: Ще дефинираме класа prat, като член-данна в него е обект на класа rat.

```
class prat
{public:
  prat(int, int, int);
  ...
 private:
  int a;
  rat r;
};
```

където

```
prat::prat(int x, int y, int z)
{a = x;
 r = rat(y, z);
}
```

и сме дефинирали обекта q

```
prat q=prat(1,2,3);
```

Преди да започне изпълнението на конструктора prat, автоматично се извиква конструкторът по подразбиране на rat и член-данната r на prat се инициализира с 0/1. Веднага след това r се свързва с обекта rat(y,z) за текущите y и z. По-ефективно е данната r да се свърже с правилната стойност направо, без междинна инициализация. Това може да се реализира чрез дефиницията:

```
prat::prat(int x, int y, int z): r(rat(y, z))
{a = x;
}
```

или съкратено

```
prat::prat(int x, int y, int z): r(y, z)
{a = x;
}
```

14.4.2 Предефинирани конструктори

Обект (в общия смисъл) е предефиниран, ако за него има няколко различни дефиниции, задаващи различни негови интерпретации. За да

бъдат използвани такива конструкции е необходим критерии, по който те да се различат.

В рамките на една програма може да се извършва предефиниране на функции. Възможно е:

а) да се използват функции с едно и също име с различни области на видимост

В този случай не възниква проблем с различаването.

б) да се използват функции с едно и също име в една и съща област на видимост

В този случай компилаторът търси функцията с възможно най-добро съвпадане. Като критерии за добро съвпадане са въведени следните нива на съответствие:

- точно съответствие (по брой и тип на формалните и фактическите параметри)
- съответствие чрез разширяване на типа.
  Извършва се разширяване по веригата
  char -> short -> int -> longint или
  float -> double
- други съответствия (правила въведени от потребителя).

В един клас може да са дефинирани няколко конструктора. Всички те имат едно име (името на класа), но трябва да се различават по броя и/или типа на параметрите си. Наричат се **предефинирани конструктори**. При създаването на обект на класа се изпълнява само един от тях. Определя се съгласно критерия за най-добро съвпадане.

Пример: В класа rat дефинирахме два конструктора rat() и rat(int, int), които се различават по броя на параметрите си.

14.4.3 Подразбиращ се конструктор

В клас може явно да е дефиниран, но може и да не е дефиниран конструктор. Ако явно не е дефиниран конструктор, автоматично се създава един т. нар. **подразбиращ се конструктор**. Този конструктор реализира множество от действия като: заделяне на памет за данните на обект, инициализиране на някои системни променливи и др.

Подразбиращият се конструктор може да бъде предефиниран. За целта е необходимо в класа да бъде дефиниран конструктор без параметри.

Пример: в класа rat, подразбиращият се конструктор беше предефиниран от конструктора

```
rat::rat()
{numer = 0;
 denom = 1;
}
```

14.4.4 Конструктори с подразбиращи се параметри

Функциите в езика C++ могат да имат подразбиращи се параметри. За тези параметри се задават подразбиращи се стойности, които се използват само ако при извикването на функцията не бъде зададена стойност за съответния параметър.

Задаването на подразбираща се стойност се извършва чрез задаване на конкретна стойност в прототипа на функцията или в нейната дефиниция.

Пример: Да разгледаме програмата

```
#include <iostream.h>
void f(double, int=10, char* ="example1"); //интервал между * и
=
int main()
{double x = 1.5;
 int y = 5;
 char z[] = "example 2";
 f(x, y, z);
 f(x, y);
 f(x);
 return 0;
}
void f(double x, int y, char* z)
{cout << "x= " << x << " y= " << y
      << " z= " << z << endl;
}
```

В тази програма е дефинирана функцията f с три формални параметъра. От прототипа й се вижда, че два от тях (вторият и третият) са подразбиращи се със стойности по подразбиране 10 и “example 1” съответно. Тъй като в обръщенията към f

```
f(x, y); и
```

```
f(x);
```
са указани по-малко от три фактически параметъра, за стойности на липсващите параметри се вземат указаните стойности от прототипа на функцията.

В резултат от изпълнението на програмата се получава:

```
x = 1.5 y = 5 example 2
x = 1.5 y = 5 example 1
x = 1.5 y = 10 example 1
```

При използване на подразбиращи се параметри, важна роля играе редът на параметрите. Прието е, че ако параметър на функция е подразбиращ се, всички параметри след него също са подразбиращи се.

Пример: Прототип на функция

```
void f(double = 1.5, int, char* “example 1”);
```
предизвиква грешка, тъй като първият формален параметър е обявен за подразбиращ се, а вторият не е такъв.

Ще отбележим също, че ако за даден подразбиращ се параметър е зададена стойност при обръщението към функцията, за всички параметри *пред него* също трябва да са указани такива.

Всичко казано за подразбиращите се параметри на функции се отнася и за конструкторите.

Ако променим дефиницията на класа rat по следния начин:

```
class rat
{private:
  int numer;
  int denom;
 public:
  rat(int=0, int=1);
  void read();
  int get_numer() const;
  int get_denom() const;
  void print() const;
};
```
където конструкторът rat(int, int); се дефинира по същия начин, са допустими следните дефиниции на обекти:

```
rat p,                  // p се инициализира с 0/1
    q = rat(),          // q се инициализира с 0/1
    r = rat(5),         // r се инициализира с 5/1
```

```
        s = rat(13,21),     // s се инициализира с 13/21
        t(2);               // t се инициализира с 2/1
```

14.4.5 Конструктори за присвояване и копиране

Инициализацията на новосъздаден обект на даден клас може да зависи от друг обект на същия клас. За да се укаже такава зависимост се използва знакът за присвояване или кръгли скоби.

Пример: Нека

```
rat p = rat(1,4);
```

Чрез еквивалентните конструкции

```
rat q = p;
rat q(p);
```

се създава обектът q от клас rat, като инициализацията на q зависи от p. Тази инициализация се създава от специален конструктор, наречен **конструктор за присвояване**.

Конструкторът за присвояване е конструктор, поддържащ формален параметър от тип <име_на_клас> const &.

- Ако в един клас явно не е дефиниран конструктор за присвояване, компилаторът автоматично създава такъв, в момента когато новосъздаден обект се инициализира с обекта, намиращ се от дясната страна на знака за присвояване или в кръглите скоби. Този конструктор за присвояване се нарича конструктор за копиране.

Пример: В класа rat не беше дефиниран конструктор за присвояване. Затова при създаване на обект p чрез дефиницията rat p = q;
автоматично се извиква конструкторът за копиране. Последният има вида:

```
rat::rat(rat const & r)
{numer = r.numer;
 denom = r.denom;
}
```

или по-точно

```
rat::rat(rat* this, rat const & r)
{this -> numer = r.numer;
 this -> denom = r.denom;
}
```

Дефиницията rat p=q създава нов обект p *(без викане на конструктор)*, в който се копират съответните стойности на обекта q. Това е резултат от изпълнението на обръщението към rat(&p, q).

- Ако в класа е дефиниран конструктор за присвояване, компилаторът го използва.

Пример: Ще добавим един безсмислен, даже глупав, конструктор за присвояване към класа rat. Правим го с експериментална цел.

```
class rat
{private:
  int numer;
  int denom;
 public:
  rat(rat const &); // конструктор за присвояване
  rat();
  rat(int=0, int=1);
  void read();
  int get_numer() const;
  int get_denom() const;
  void print() const;
};
...
rat::rat(rat const & r)
{numer = r.numer + 1;
 denom = r.denom + 1;
}
```

Компилаторът превежда този конструктор за присвояване във вида:

```
rat_rat(rat *this, rat const &r)
{this->numer = r.numer + 1;
 this->denom = r.denom + 1;
}
```

а

```
rat q = p;
```

в

```
rat_rat(&q, p);
```

Така дефинираният конструктор за присвояване увеличава с 1 числителя и знаменателя на рационалното число p, фактически параметър на обекта r (формален параметър на конструктора) и ги

свързва с числителя и знаменателя на обекта q сочен от указателя this.

В резултат имаме:

```
rat p,         // p се инициализира с 0/1
    q = p,     // q се инициализира с 1/2
    r = q      // r се инициализира с 2/3
    s = r,     // s се инициализира с 3/4
    t(s);      // t се инициализира с 4/5.
```

Предефиниране на служебния конструктор за копиране се налага когато обектите имат член-данни, указващи динамична памет.

Освен в горните случаи, конструкторът за присвояване (копиране) се използва и при предаване на обект като аргумент на функция, когато предаването е по стойност, а също при връщане на обект като резултат от изпълнение на функция.

Правилата, определящи достъпа на функциите до обектите, съществено не се различават от тези, регламентиращи достъпа на функциите до обикновените променливи. Обектите могат да се предават като параметри на функциите по един от известните вече три начина: по стойност, по указател и по псевдоним. При предаване по стойност функциите работят с *копия* на параметрите, а не със самите параметри. При другите два начина за предаване на параметрите не се правят копия (функциите работят с оригиналните параметри).

Когато обектите се предават като параметри на функции, спецификаторите на достъп public и private имат същия смисъл, т.е. функциите имат достъп само до public компонентите на обектите, когато им се подават като параметри.

Пример: Нека останем в означенията на класа rat с глупавия конструктор за присвояване от предишния пример и нека функцията sum, намираща сума на две рационални числа, е дефинирана по следния начин:

```
a) rat sum(rat r1, rat r2)
   {rat r(r1.get_numer()*r2.get_denom()+
          r2.get_numer()*r1.get_denom(),
          r1.get_denom()*r2.get_denom());
    return r;
   }
```

Обръщението sum(p, q).print() извежда 8/7. Този резултат може да се обясни по следния начин. Тъй като r1 и r2 са параметри стойности, те се свързват с фактическите си параметри чрез присвояване. В резултат r1 се свързва с ½ (не с 0/1), а r2 – с 2/3 (не с ½). След изпълнението на инициализацията

```
rat r(r1.get_numer()*r2.get_denom()+
       r2.get_numer()*r1.get_denom(),
       r1.get_denom()*r2.get_denom());
```

/чрез двуаргументния конструктор rat(int, int)/ обектът r се свързва със 7/6. Тъй като функцията rat е от тип rat, при изпълнение на return r; се прави още едно прилагане на глупавото присвояване и се получава 8/7.

б)
```
rat sum(rat const & r1, rat const & r2)
   {rat r(r1.get_numer()*r2.get_denom()+
          r2.get_numer()*r1.get_denom(),
          r1.get_denom()*r2.get_denom());
    return r;
   }
```

Сега обръщението sum(p, q).print() извежда 2/3. Този резултат може да се обясни по следния начин. Тъй като r1 и r2 са параметри псевдоними, те директно се свързват с фактическите си параметри (не се извършва присвояване), т.е. r1 се свързва с 0/1, а r2 – 1/2. След изпълнението на инициализацията

```
rat r(r1.get_numer()*r2.get_denom()+
       r2.get_numer()*r1.get_denom(),
       r1.get_denom()*r2.get_denom());
```

r се свързва със 1/2. Аналогично на случай а), при изпълнение на return r; се прилагане “глупавото” присвояване и се получава 2/3.

в)
```
rat sum(rat* r1, rat* r2)
   {rat r(r1->get_numer()*r2->get_denom()+
          r2->get_numer()*r1->get_denom(),
          r1->get_denom()*r2->get_denom());
   return r;
   }
```

и

```
rat* p1 = &p,
   * q1 = &q;
```

Обръщението sum(p1, q1).print() извежда 2/3. Този резултат може да се обясни по следния начин. Тъй като r1 и r2 са параметри указатели, те се свързват с фактическите си параметри чрез адрес. В резултат r1 се свързва с 0/1, а r2 – с 1/2. След изпълнението на инициализацията

```
rat r(r1->get_numer()*r2->get_denom()+
      r2->get_numer()*r1->get_denom(),
      r1->get_denom()*r2->get_denom());
```

r се свързва със 1/2. Тъй като функцията rat е от тип rat, при изпълнение на return r; се прилага присвояването и се получава 2/3.

14.5 Указатели към обекти на класове

Дефинират се по същия начин както се дефинират указатели към променливи от стандартните типове данни.

Пример:

```
rat p;
rat * ptr = &p;
```

В резултат ще се отделят 4В ОП, които ще се именуват с ptr и ще се инициализират с адреса на обекта p.

Достъпът до компонентите на рационалното число, сочено от ptr, се осъществява по общоприетия начин:

```
(*ptr).get_numer()
(*ptr).get_denom()
```

Синтактичната конструкция (*ptr). е еквивалентна на ptr ->. Така горните обръщения могат да се запишат и по следния начин:

```
ptr -> get_numer()
ptr -> get_denom()
```

Ще напомним, че this е указател от тип <име_на_клас>*.

14.6 Масиви и обекти

Елементите на масив могат да са обекти, но разбира се от един и същ клас. Дефинират се по общоприетия начин (Фиг. 14.7).

Дефиниция на масив от обекти

<дефиниция_на_променлива_от_тип_масив_от_обекти> ::=

```
     T <променлива>[size] [= {<инициализиращ_списък>}]опц;
```
където

- T е име или декларация на клас;
- <променлива> е идентификатор;
- size е константен израз от интегрален или изброен тип с *положителна* стойност;
- <инициализиращ_списък> се дефинира по следния начин:

```
<инициализиращ_списък> ::= <стойност>{, <стойност>}опц
      {, <име_на_конструктор>(<фактически_параметри>)}опц
```

Фиг. 14.7 Дефиниция на масив от обекти

Пример:

```
rat table[10];
```

определя масив от 10 обекта от клас rat.

Достъпът до елементите на масива е пряк и се осъществява по стандартния начин – чрез индексирани променливи.

Пример: Чрез индексираните променливи

```
table[0], table[1], ..., table[9]
```

се осъществява достъп до първия, втория и т.н. до десетия елемент на table.

Тъй като table[i] (i = 0, 1, ..., 9) са обекти, възможни са следните обръщения към техни компоненти:

```
table[i].read();        // въвежда стойност на table[i]
table[i].print();       // извежда стойността на table[i]
table[i].get_numer();   // намира числителя на table[i]
table[i].get_denom();   // намира знаменателя на table[i].
```

Връзката между масиви и указатели е в сила и в случая когато елементите на масива са обекти. Името на масива е указател към първия му елемент, т.е. ако

```
rat * p = table;     // p сочи към table[0]
                     // т.е. p==&table[0]
```

*(p+i) == table[i], i = 0, 1,...,9

Тогава

```
(*(p+i)).print();    // е еквивалентно на table[i].print();
```

Масивът може да е член-данна на клас.

Пример: Конструкцията

```
class example
```

```
{int a;
 int table[10];
 public:
 int array[10];
} x[5];
```

дефинира масив с 5 компоненти, които са от тип example. Достъпът до компонентите на масива array ще се осъществи по следния начин:

x[i].array[j], i = 0, 1, ..., 4; j = 0, 1, ..., 9.

Конструкторите (в частност конструкторът по подразбиране) играят важна роля при дефинирането и инициализирането на масиви от обекти. Масив от обекти, дефиниран в програма, се инициализира по два начина:

- *неявно* (чрез извикване на системния конструктор по подразбиране за всеки обект – елемент на масива);
- *явно* (чрез инициализиращ списък).

Примери:

а) Класът

```
const NUM = 5;
class student
{public:
 void read_student();
 void print_student() const;
 bool is_better(student const &) const;
 double average() const;
 private:
 int facnom;
 char name[26];
 double marks[NUM];
};
```

няма явно дефиниран конструктор. Дефиницията

```
student table[30];
```

на масива table от 30 обекта от клас student е правилна. Инициализацията се осъществява чрез извикване на "системния" конструктор по подразбиране за всеки обект – елемент на масива.

б) Класът rat, дефиниран по-долу

```
class rat
{private:
  int numer;
```

```
    int denom;
   public:
    rat(int=0, int=1);
    void read();
    int get_numer() const;
    int get_denom() const;
    void print() const;
  };
```

притежава явно дефиниран конструктор с два подразбиращи се параметъра. В този случай са допустими дефиниции от вида:

```
rat x[10]; // x[i] се инициализира с 0/1, за всяко i=0,1,...9.
rat x[10] = {1,2,3,4,5,6,7,8,9,10}; //x[i] == i/1
rat x[10] = {rat(1,21),rat(2),rat(3, 5),4,5,6,7,8,9,10};
// x[0] == 1/21; x[1] == 2/1; x[2] == 3/5, x[3]== 4/1, ...
```

Ако променим конструктора на класа rat от

```
rat(int=0, int=1);
```

в

```
rat(int, int);
```

т.е. без подразбиращи се параметри и трите дефиниции от по-горе ще съобщят за грешка. Единствено допустима дефиниция на x[10] е с инициализация с 10 обръщения към двуаргументния конструктор rat с явно указани два аргумента.

Задача 120. Да се напише програма, която въвежда следната информация за компютри: име на модела (name), цена (price) и точки (score) между 1 и 100. Да се изведе въведената информация, след което да се изведе сортирана в низходящ ред по съотношението точки/цена.

Отново ще използваме подхода абстракция със структури от данни. Първите две нива на абстракция ще реализираме чрез дефиницията на класа

```
class product
{public:
  void read();
  void print() const;
  bool is_better_from(product const &) const;
  double get_price() const;
```

```
    int get_score() const;
   private:
    char name[21];
    double price;
    int score;
  };
```

Примитивните операции се реализирани чрез следните член-функции:

```
  void read();            - въвежда информация за компютър
  void print() const;     - извежда информация за компютър
  bool is_better_from(product const &) const;
                          -  проверява дали текущият компютър има
                             по-добро съотношение score/price
                             от това на указания като формален
                             параметър
  double get_price() const; - намира цената на компютър
  int get_score() const;   - намира точките на компютър
```

Програма Zad120.cpp решава задачата.

```
  // Program Zad120.cpp
  #include <iostream.h>
  #include <iomanip.h>
  #include <string.h>
  class product
  {private:
    char name[21];
    double price;
    int score;
   public:
    void read();
    void print() const;
    bool is_better_from(product const &) const;
    double get_price() const;
    int get_score() const;
  };
  void sorttable(int n, product* []);
  int main()
  {cout << setprecision(4) << setiosflags(ios::fixed);
```

```
 product table[300];
 product* ptable[300];
 int n;
 do
 {cout << "number of products? ";
  cin >> n;
 } while (n<1 || n>300);
 int i;
 for (i = 0; i <= n-1; i++)
 {table[i].read();
  ptable[i] = &table[i];
 }
 cout << "table: \n";
 for (i = 0; i <= n-1; i++)
 {table[i].print();
  cout << endl;
 }
 sorttable(n, ptable);
 cout << "\n New Table: \n";
 for (i = 0; i <= n-1; i++)
 {ptable[i]->print();
  cout << setw(7)
       << ptable[i]->get_score()/ptable[i]->get_price()
       << endl;
 }
 return 0;
}
void product::read()
{cout << "name: ";
 cin >> name;
 cout << "price: ";
 cin >> price;
 cout << "score: ";
 cin >> score;
}
void product::print() const
{cout << setw(25) << name
      << setw(10) << price
```

```
      << setw(12) << score;
}
bool product::is_better_from(product const & x) const
{return score/price > x.score/x.price;
}
double product::get_price() const
{return price;
}
int product::get_score() const
{return score;
}
void sorttable(int n, product* a[])
{for (int i = 0; i <= n-2; i++)
 {int k = i;
  product* max = a[i];
  for (int j = i+1; j <= n-1; j++)
  if (a[j]->is_better_from(*max))
   {max = a[j];
    k = j;
   }
  max = a[i]; a[i] = a[k]; a[k] = max;
 }
}
```

Ще дадем още един пример, показващ връзка между класове и масиви. В него масивът е член-данна на клас, описващ последователно представяне на структурата от данни стек.

14.7 Стек

Стекът е линейна динамична структура от данни. В Глава 8 (Увод в програмирането на базата на езика C++) направихме кратко описание на тази структура. В тази част ще направим по-пълно описание. Ще започнем със следната задача.

Задача 121. Да се напише програма, която извежда двоичното представяне на естествено число.

Операторът

```
do
```

```
{cout << k%2;
 k/=2;
} while (k);
```

извежда двоичното представяне на числото k, но в обратен ред. За решаване на задачата ще използваме динамичната структура от данни **стек**.

Стекът е крайна редица от елементи от един и същ тип. Операциите включване и изключване на елемент са допустими само за единия край на редицата, който се нарича **връх на стека**. Възможен е достъп само до елемента, намиращ се на върха на стека като достъпът е пряк.

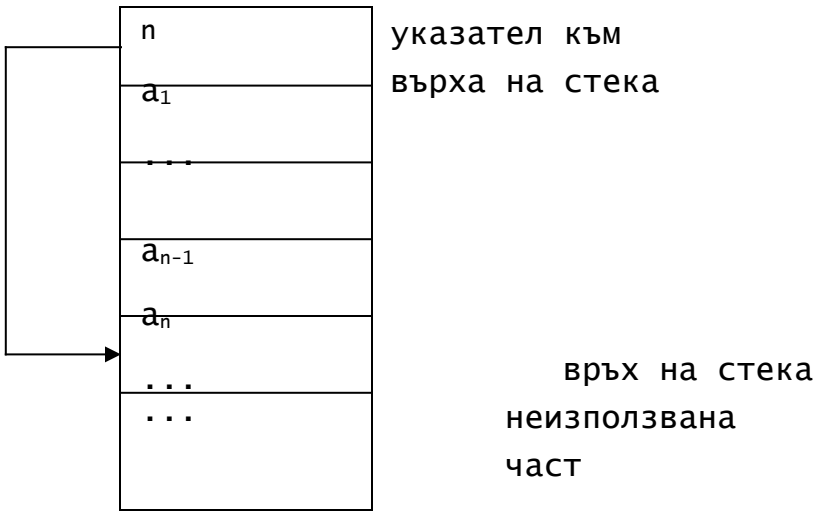
При тази организация на логическите операции, последният включен елемент се изключва пръв. Затова стекът се определя още като структура "*последен влязъл – пръв излязъл*".

Широко се използват два основни начина за физическо представяне (представяне в ОП) на стек: *последователно* и *свързано*. За целите на тази задача ще използваме последователното представяне.

При това представяне се запазва блок от паметта, вътре в който стекът да расте и да се съкращава. Ако редицата от елементи от един и същ тип

$a_n, a_{n-1}, ..., a_1$

е стек с връх a_n, последователното представяне на стека има вида:



При включване на елементи в стека, те се поместват в последователни адреси в неизползваната част веднага след върха на стека.

Това физическо представяне ще реализираме като използваме структурата от дании масив. За указател към върха на стека ще служи цяла променлива. Ще използваме подхода абстракция със структури от данни. Първите две нива на абстракция реализираме чрез класа stack:

```
class stack
{public:
  stack();
  void push(int);
  void pop();
  void print();
  int top() const;
  bool empty() const;
 private:
  int n;    // указател към върха на стека
  int arr[NUM];   // представяне на стека
};
```

където

```
const int NUM = 100;
```

По такъв начин ограничаваме размера на стека до 99 (arr[0] ще инициализираме с 0 и няма да използваме). Масивът arr ще представя стека, а n ще е указателя към върха му.

Примитивните операции са реализирани чрез следните член-функции:

```
void push(int);          - включва елемент в стека
void pop();              - изключва елемент от стека
void print();            - извежда елементите на стека като
                           разрушава стека
int top() const;         - намира елемента от върха на стека
bool empty() const;      - проверява дали стекът е празен
```

Програма Zad121.cpp решава задачата.

```
// Program Zad121.cpp
#include <iostream.h>
const NUM = 100;
class stack
{public:
  stack();
```

```
     void push(int);
     void pop();
     int top() const;
     bool empty() const;
     void print();
    private:
     int n;
     int arr[NUM];
   };
   stack   num_stack(int);   //   конструира   стек   от   двоичното
представяне
                          // на указано цяло число
   void main()
   {cout << "number: ";
    int n;
    cin >> n;
    num_stack(n).print();
   }
   stack::stack()
   {n = 0;
    arr[0]=0;
   }
   void stack::push(int x)
   {n++;
    arr[n] = x;
   }
   void stack::pop()
   {n--;
   }
   int stack::top() const
   {return arr[n];
   }
   bool stack::empty() const
   {return n == 0;
   }
   void stack::print()
   {while (!empty())
    {cout << top();
```

```
  pop();
 }
 cout << endl;
}
stack num_stack(int x)
{stack st;
 while (x)
 {st.push(x%2);
  x/=2;
 }
 return st;
}
```

14.8 Динамични обекти

Вече разгледахме в най-общ план разпределението на ОП по време на изпълнението на програма. От Фиг. 8.1 на Глава 8 се вижда, че всяка програма има две "места" за памет: *програмен стек (стек)* и *област за динамичните данни (динамична памет, хийп или куп).*

Стекът е област за временно съхранение на информация. Той е кратковременна памет. C++ използва стека основно за реализиране на обръщения към функции. Всяко обръщение към функция предизвиква конструиране на нова стекова рамка, която се установява на върха на стека. По такъв начин когато функция A извика функция B, която от своя страна вика функция C, стекът нараства. Когато пък всяка от тези функции завършва, стековите рамки на тези функции автоматично се разрушава. Така стекът се свива.

Хийпът е по-постоянна област за съхранение на данни. Той е един вид дълготрайна памет. Особеност на тази памет е, че тя не се свързва с имена на променливи. С разположените в нея обекти се работи косвено – чрез указатели. Обикновено се използва при работа с т. нар. **динамични структури от данни**. *Динамичните данни са такива обекти (в широкия смисъл на думата), чийто брой не е известен в момента на проектирането на програмата. Те се създават и разрушават по време на изпълнението на програмата. След разрушаването им, заетата от тях памет се освобождава и може да се използва отново. Така паметта се използва по-ефективно.*

Използването на динамичната памет досега не се налагаше, тъй като структурите от данни, с които работехме, бяха статични. За целите на следващите разглеждания, когато ще дефинираме и използваме динамичните структури от данни свързан списък, стек, опашка, дърво, граф и др., използването на тези средства е задължително.

Създаването и разрушаването на динамични обекти в C++ се осъществява чрез операторите new и delete. Извикването на new заделя в хийпа необходимата памет и връща указател към нея. Този указател може да се съхрани в някаква променлива и да се пази докато е необходимо. За разлика от стека, заделянето на памет в хийпа е явно – чрез new. Освобождаването на паметта от хийпа също става явно, чрез delete. Всяко извикване на new трябва да бъде балансирано чрез извикване на delete. Последното се налага, тъй като за разлика от стека, хийпът не се изчиства автоматично. В C++ няма система за "събиране на боклуци" (автоматично премахване на обекти, които вече не са необходими). Затова трябва явно да бъдат изтрити създадените в хийпа обекти. Описанието на оператора new е дадено на Фиг. 14.8.

Оператор new

Синтаксис

new <име_на_тип> [[size]]$_{опц}$;|

new <име_на_тип> (<инициализация>);

където

- *<име_на_тип>* е име на някой от стандартните типове int, double, char и др. или е име на клас;
- *size* е израз с произволна сложност, но трябва да може да се преобразува до цял. Показва броя на компонентите от тип <име_на_тип>, за които да се задели памет в хийпа и се нарича **размерност**;
- *<инициализация>* е израз от тип <име_на_тип> или инициализация на обект според синтаксиса на конструктора на класа, ако <име_на_тип> е име на клас.

Семантика

Заделя в хийпа (ако е възможно):

- sizeof(<име_на_тип>) байта, ако не са зададени size и <инициализация> или
- sizeof(<име_на_тип>)*size байта, ако явно е указан size или

- sizeof(<име_на_тип>) байта, ако е специфицирана <инициализация>, която памет се инициализира с <инициализация> и връща указател към заделената памет.
Ако няма достатъчно памет в хийпа, операторът new връща NULL.

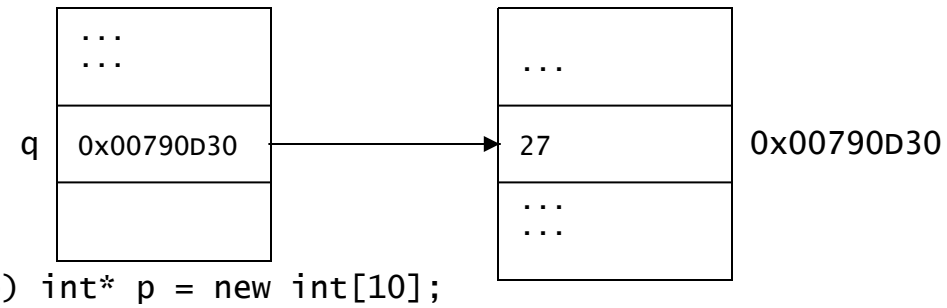
Фиг. 14.8 Оператор new

Забележка: Ако <име_на_тип> е име на клас и след него има кръгли скоби, в тях трябва да стоят фактически параметри (аргументи) на конструктора на класа. Ако скобите липсват, класът трябва да притежава конструктор по подразбиране или да няма явно дефиниран конструктор. Ако след името на класа е поставен заграден в квадратни скоби израз, new заделя място за масив от обекти на указания клас и извиква конструктора по подразбиране за инициализиране на отделената памет.

Примери:

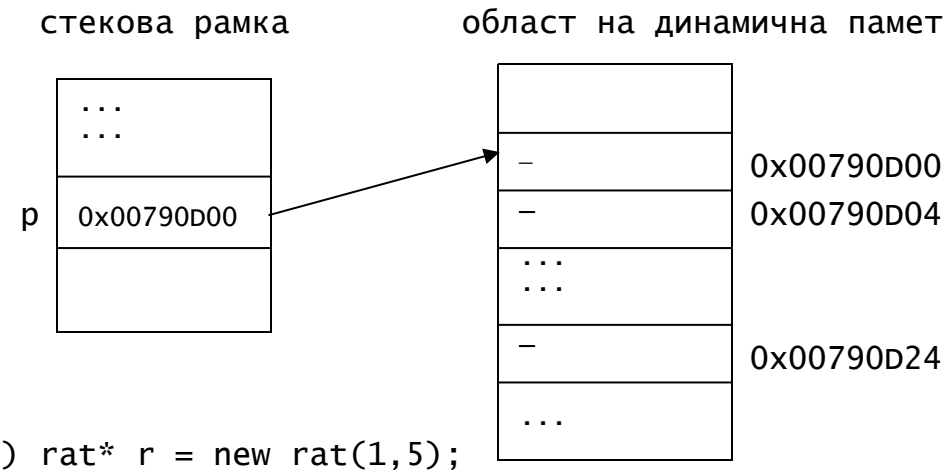
а) int* q = new int(2+5*5);

отделя (ако е възможно) 4B памет в хийпа, инициализира я с 27 – стойността на израза 2+5*5 и свързва q с адреса на тази памет, т.е.



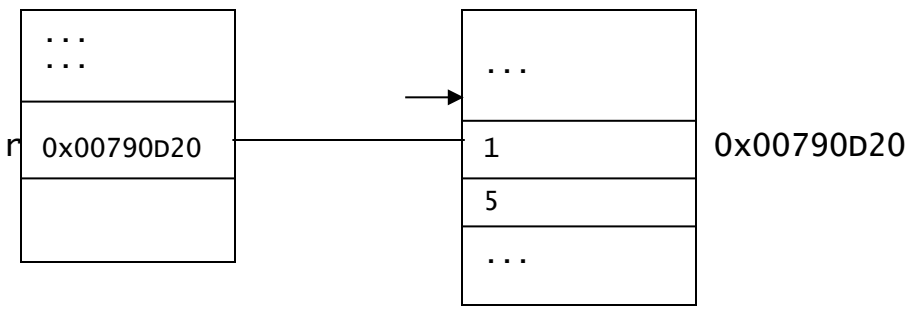
б) int* p = new int[10];

отделя (ако е възможно) 40B в хийпа (за 10 елемента от тип int) и свързва p с адреса на тази памет, т.е.



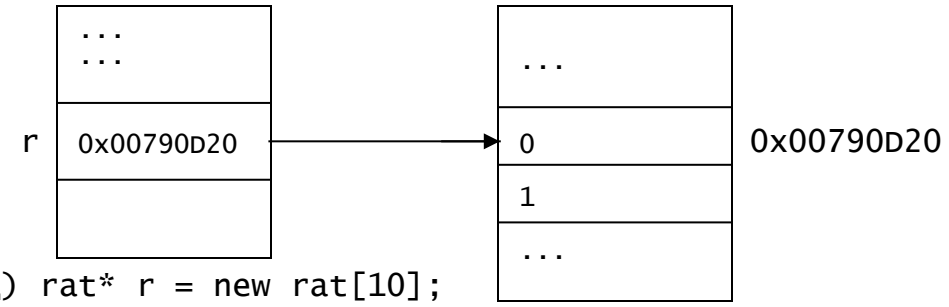
в) rat* r = new rat(1,5);

отделя памет в хийпа за един обект от клас rat, свързва r с адреса на тази памет и извиква конструктора rat(1,5) за да я инициализира, т.е.



г) rat* r = new rat;

отделя памет в хийпа за обект от тип rat, записва адреса на тази памет в r и извиква конструктора по подразбиране на класа rat за инициализиране на тази памет, т.е.

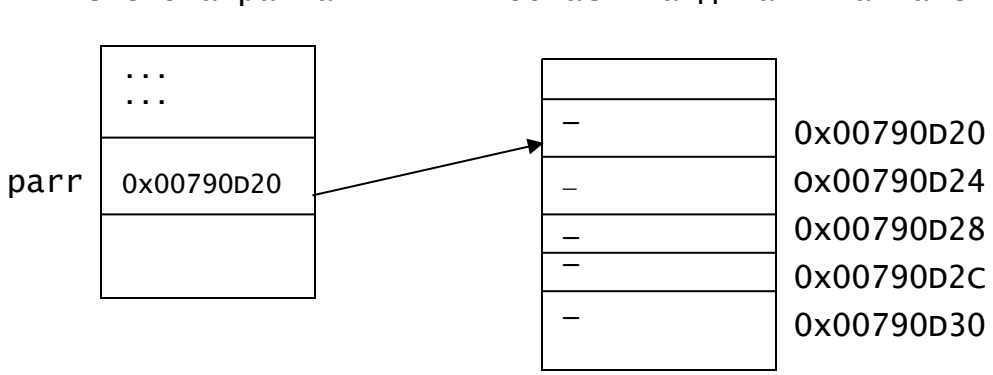


д) rat* r = new rat[10];

отделя памет в хийпа за 10 обекта от класа rat, записва адреса на тази памет в r, извиква конструктора по подразбиране на класа rat и инициализира отделената памет;

е) rat** parr = new rat*[5];

отделя 20в памет в хийпа за масив от 5 указателя към стойности от тип rat и записва в parr адреса на тази памет, т.е.



Заделянето на памет по време на компилация се нарича **статично** заделяне на памет, заделянето на памет по време на изпълнение на програмата - **динамично разпределение на паметта**. Паметта за променливите q, p, r и parr, от примерите по-горе, е заделена статично, а всяка една от тези променливи има за стойност адрес от хийпа. Казва се още, че q, p, r и parr адресират динамична памет.

Под период на **активност на една променлива** се разбира частта от времето за изпълнение на програмата, през което променливата е свързана с конкретно място в паметта. Паметта за глобалните променливи се заделя в началото и остава свързана с тях до завършването на изпълнението на програмата. Паметта за локалните променливи се заделя при влизане в локалната област и се освобождава при напускането ѝ. Паметта на динамичните променливи се заделя от оператора new. Заделената по този начин памет остава свързана със съответната променлива докато не се освободи явно от програмиста. Явното освобождаване на динамична променлива се осъществява чрез оператора delete, приложен към указателя, който адресира съответната променлива. Едно непълно негово описание е дадено на Фиг. 14.9.

Оператор delete

Синтаксис

delete <указател_към_динамичен_обект>;

където <указател_към_динамичен_обект> е указател към динамичен обект (в широкия смисъл на думата), създаден чрез оператора new.

Семантика

Разрушава обекта, адресиран от указателя, като паметта, която заема този обект, се освобождава. Ако обектът, адресиран от указателя, е обект на клас, отначало се извиква деструкторът на класа (т.15.9) и след това се освобождава паметта.

Фиг. 14.9 Оператор delete

Ако в хийпа е заделена памет, след което тази памет не е освободена чрез delete, се получава загуба на памет. Парчето памет, което не е освободено, е като остров в хийпа, заемащо пространство, което иначе би могло да се използва за други цели.

За да се разруши масив, създаден чрез new по следния начин:

```
int* arr = new int[5];
```

трябва да се запише:

```
delete [] arr;
```

Забележка: Някои реализиции на езика допускат разрушаването в горния случай да стане и чрез delete arr.

Ако обаче масивът съдържа в себе си указатели, първо трябва да бъде обходен и да бъде извикан операторът delete за всеки негов елемент. Едва след това може да се извика delete за масива по показания по-горе начин.

Следващата задача илюстрира казаното.

Задача 122. Да се напише програма, която отделя динамична памет за масив от 10 указателя към тип int. Програмата да проверява дали отделянето на памет е станало; запълва масива с указатели към цели числа; извежда стойностите и съдържанието на указателите и освобождава отделената динамична памет.

```
// Program Zad122.cpp
#include <iostream.h>
int main()
{// заделяне на памет за масив от указатели към int
 int**  arr = new int*[10];
 if (arr == NULL)  //или if (!arr)
 {cout << "Not enough memory!\n";
  return 1;
 }
 // запълване на масива с указатели
 int i;
 for (i=0; i<10; i++)
 {arr[i] = new int;     // заделяне на памет за указателя
  if (arr[i]==NULL) //или if (!arr[i])
   {cout << “Not enough memory!\n”;
    return 1;}
  *arr[i] = i;    // инициализиране на указваната стойност
 }
 // извеждане на стойностите и съдържанието на указателите
 for (i=0; i<10; i++)
```

```
      cout << arr[i] << “  “ << *arr[i] << ", ";
    cout << endl;
    // освобождаване на заетата динамична памет
    for (i = 0; i < 10; i++)
      delete arr[i];
    delete [] arr;
    return 0;
   }
```

Операторът delete трябва да се използва само за освобождаване на динамична памет, заделена с new. В противен случай действието му е непредсказуемо. Няма забрана за прилагане на delete към указател със стойност 0. Ако стойността на указателя е 0, той е свободен и не адресира нищо.

Пример:

```
void example()
{...
 int a = 7;
 char *str = “abv”;
 int *pa = &a;
 rat *ptr = 0;
 double *x = new double;
  delete str; //некоректно обръщение, str не е адресирано чрез new
 delete pa; //некоректно обръщение, pa не е адресирано чрез new
  delete ptr; //некоректно обръщение, ptr не е адресирано чрез new
 delete x; // коректно обръщение
 ...
}
```

Динамичната памет не е неограничена. Тя може да се изчерпи по време на изпълнение на програмата. Неуспешното завършване на delete ускорява изчерпването. Ако наличната в момента динамична памет е недостатъчна, new връща нулев указател и програмата (функцията) няма да работи. Затова се препоръчва след всяко извикване на new да се прави проверка за успешността ѝ.

Чрез оператора new могат да се създават т. нар. **динамични масиви** – масиви с променлива дължина. Динамичните масиви се

създават в динамичната памет. Следващата програма илюстрира този процес.

Задача 123. Да се напише програма, която създава динамичен масив от цели числа. Да се изведе масивът.

```
// Program 123. cpp
#include <iostream.h>
int main()
{int size; // дължина на масива
 do
 {cout << "size of array: ";
  cin >> size;
 } while (size<1);
 // създаване на динамичен масив arr от size елемента от тип int
 int* arr = new int[size];
 int i;
 for (i = 0; i <= size-1; i++)
  arr[i] = i;
 // извеждане на елементите на arr
 for (i = 0; i <= size-1; i++)
  cout << arr[i] << " ";
 cout << endl;
 // освобождаване на заетата динамична памет
 delete [] arr;
 return 0;
}
```

Чрез оператора

```
delete [] arr;
```

е разрушен динамичният масив arr. Това може да стане и ако се напише само

```
delete arr;
```

тъй като елементите на масива arr са числа.

Забелязваме, че размерът size на динамичния масив може да се въведе или изчисли по време на изпълнение на програмата и не е задължително да бъде известен по време на компилация. Това

позволява да се подобри програмата на задача 120, като се използват динамични, а не “статични” масиви (т. 14.8).

Методите на класовете също могат да използват динамична памет, която се заделя и освобождава по време на изпълнението им, чрез операторите new и delete.

Задача 124. Да се модифицира класът product, реализиран в задача 120, така че за всяко име на компютър да се отделя точно толкова памет, колкото е необходимо, а не точно 21 байта.

За целта ще определим променливата name като указател към char и необходимата памет за съхраняването на името на компютър ще се заделя по време на изпълнение на член-функцията read(). Следват само фрагментите, където се налага модификация.

```
char s[40];
class product
{private:
  char* name;
  double price;
  int score;
 public:
  void read();
  void print() const;
  bool is_better_from(product const &) const;
  double get_price() const;
  int get_score() const;
};
...
void product::read()
{cout << "name: ";
 cin >> s;
 name = new char[strlen(s)+1];
 strcpy(name, s);
 cout << "price: ";
 cin >> price;
 cout << "score: ";
 cin >> score;
}
```

Използвана е глобална променлива s от тип низ, която играе ролята на буфер – временно съхранява въведено име. След това се намира дължината на този низ и в динамичната памет за name се отделя точно толкова памет, колкото е необходимо. Така за член-данната name на всеки обект на класа product ще се заделя нужната памет, а не винаги 21 B, която памет може да се окаже и недостатъчна.

Ще отбележим също, че заделената от член-функциите динамична памет не се освобождава автоматично при разрушаване на обектите на класове. Освобождаването на тази памет трябва да стане явно чрез оператора delete, който трябва да се изпълни преди разрушаването на обекта. Този процес може да бъде автоматизиран чрез използване на специален вид методи, наречени **деструктори**.

14.9 Деструктори

Разрушаването на обекти на класове в някои случаи е свързано с извършване на определени действия, които се наричат **заключителни**. Най-често тези действия са свързани с освобождаване на заделена преди това динамична памет, възстановяване на състояние на програмата и др. Ефектът от заключителните действия е противоположен на ефекта на инициализацията. Естествено е да се даде възможност заключителните действия да се извършат автоматично при разрушаването на обекта. Това се осъществява от деструкторите.

Деструкторът е член-функция, която се извиква при:
- разрушаването на обект чрез оператора delete,
- излизане от блок, в който е бил създаден обект от класа.

Един клас може да има явно дефиниран точно един деструктор. Името му съвпада с името на класа, предшествано от символа ‘~’ (тилда), типът му е void и явно не се задава в заглавието. Деструкторът няма формални параметри.

Забележка: Използването на явно дефинирани деструктори не винаги е належащо, тъй като всички член-променливи се разрушават при разрушаването на обекта и без използването на деструктор. Ако конструкторът или някоя член-функция реализира динамично заделяне на памет за някоя член-данна, използването на деструктор е задължително, тъй като в този случай той трябва да освободи заетата памет.

Задача 125. Да се промени класът product, дефиниран в програма Zad120.cpp, като методът read() се преобразува в конструктор по подразбиране и се добави деструктор. Освен това table и ptable да се реализират като динамични масиви.

Програма Zad125.cpp решава тази задача. Освен исканите модификации тя добавя и функцията за достъп

```
char* get_name() const;
```

която не е използвана в това приложение. Телата на някои методи и функции на програмата, които не са модифицирани, са пропуснати.

```
// Program Zad125.cpp
#include <iostream.h>
#include <iomanip.h>
#include <string.h>
char s[40];
class product
{private:
  char* name;
  double price;
  int score;
 public:
  product();
  ~product();
  void print() const;
  bool is_better_from(product const &) const;
  char* get_name() const;
  double get_price() const;
  int get_score() const;
};
void sorttable(int n, product* a[]);
int main()
{cout << setprecision(4) << setiosflags(ios::fixed);
 cout << "size: "; // размерност на масива
 int size;
 cin >> size;
 //създава динамичен масив от size обекта на product
 product* table = new product[size];
```

```
 // заделя памет за динамичен масив от указатели
 // към size обекта на product
 product** ptable = new product*[size];
 int i;
 cout << "table: \n";
 for (i = 0; i <= size-1; i++)
 {table[i].print();
  cout << endl;
  ptable[i] = &table[i];
 }
 sorttable(size, ptable);
 cout << "\n New Table: \n";
 for (i = 0; i <= size-1; i++)
 {ptable[i]->print();
  cout << setw(7)
       << ptable[i]->get_score()/ptable[i]->get_price()
       << endl;
 }
 delete [size] table; // някои реализации допускат
                      // пропускането на size
 delete [] ptable;      // някои реализации допускат
                        // пропускането на []
 return 0;
}
product::~product()
{delete name;
}
product::product()
{cout << "name: ";
 cin >> s;
 name = new char[strlen(s)+1];
 strcpy(name, s);
 cout << "price: ";
 cin >> price;
 cout << "score: ";
 cin >> score;
}
void product::print() const
```

```
{cout << setw(25) << name
      << setw(10) << price
      << setw(12) << score;
}
bool product::is_better_from(product const & x) const
{return score/price > x.score/x.price;
}
char* product::get_name() const
{return name;
}
double product::get_price() const
{return price;
}
int product::get_score() const
{return score;
}
void sorttable(int n, product* a[])
{for (int i = 0; i <= n-2; i++)
 {int k = i;
  product* max = a[i];
  for (int j = i+1; j <= n-1; j++)
  if (a[j]->is_better_from(*max))
   {max = a[j];
    k = j;
   }
  max = a[i]; a[i] = a[k]; a[k] = max;
 }
}
```

В резултат от изпълнението на оператора

```
product* table = new product[size];
```

се създава динамичен масив table със size обекта на класа product. При създаването на всеки от тези обекти се изпълнява конструкторът на класа, т.е. за всеки обект на table ще бъдат въведени име, цена и точки.

Операторът

```
product** ptable = new product*[size];
```

предизвиква заделяне на памет за динамичен масив от указатели към size обекта на product.

Накрая, чрез операторите

```
delete [size] table;
delete [] ptable;
```

се разрушават динамичните масиви. При разрушаването на масива table деструкторът се изпълнява size пъти, което води до освобождаване на заделената чрез конструктора памет за съхраняване на имената на обектите.

Недостатък на горната програма е, че не анализира резултата от new. Възможно е да няма достатъчно памет в хийпа. Тази ситуация е критична и изпълнението на програмата трябва да завърши. Това може да се коригира като след използването на new се добавят програмни фрагменти от вида:

```
if (!table)
{cout << "Not enough memory!\n";
 return 1;
}
```

Забележка: Ако се освобождава памет, заета от динамичен масив, чийто елементи са обекти на клас, трябва явно да се посочи дължината на масива. Тя е необходима за да се определи броят на извикванията на деструктора.

По повод на това, че за всяко обръщение към new трябва да има съответен delete, възниква въпросът: *Когато функция върне указател или псевдоним към обект, създаден чрез new, кой носи отговорността за извикването на delete за този указател?*

Например, къде в програмния фрагмент

```
struct object
{int a, b;
}
object& myfunc();
int main()
{object& rmyobj = myfunc();
 cout << rmyobj.a << rmyobj.b << endl;
 return 0;
}
object& myfunc()
{object *o = new object;
 o->a = 20;
 o->b = 40;
```

```
    return *o; // връща самия обект
  }
```

да бъде изтрит rmyobj?

Функцията, която създава указателя или псевдонима нищо не може да направи, защото когато указателят или псевдонимът бъде върнат, тя вече ще е завършила. Така че, който е извикал функцията, той след това трябва да извика и delete. Възможно е извикващият да е програма, принадлежаща на друг програмист или ваша стара програма и да не помните тази подробност. Затова като цяло *се смята за лошо програмиране връщането на указател, който после трябва да бъде изтрит. По-добре е да се върне обекта по стойност.* В примерната програма по-горе в main, след извеждането на rmyobj трябва да се включи операторът delete &rmyobj.

14.10 Създаване и разрушаване на обекти на класове

Съществуват два начина за създаване на обекти:

- чрез дефиниция;
- чрез функциите за динамично управление на паметта.

В първия случай обектът се създава при срещане на дефиницията (във функция или блок) и се разрушава при завършване на изпълнението на функцията или при излизане от блока. Дефиницията може да бъде поставена където и да е в тялото на функцията или блока, като пред и след нея може да има изпълними оператори. Дефиницията, чрез която се създава обект, може да бъде допълнена с инициализация, която може да се основава на извикване на обикновен конструктор или на конструктора за присвояване.

Разрушаването на обекта е свързано с извикването на деструктора на класа, ако такъв явно е дефиниран или с извикването на “системния” деструктор (деструктора по подразбиране), ако в класа явно не е дефиниран деструктор.

Пример: Нека в класа rat добавим деструктора

```
rat::~rat()
{cout << “destruct number: “
      << number << “/”
      << denum << endl;
}
```

Този избор на деструктор направихме с цел да наблюдаваме къде той ще бъде извикан. Ще напомним, че деструкторът извършва заключителни действия, определени от програмиста (или компилатора). Нека сега изпълним програмата с класа rat, в който е включен и деструкторът ~rat() с главна функция от вида:

```
void main()
{rat p(1,8);   // създава обект p и го инициализира с 1/8
 rat q=rat(2,9);  // създава обект q и го инициализира с 2/9
 for(int i=1; i<=5; i++)
 {rat r(i, i+1);  // създава обект r и го инициализира с i/(i+1)
  r.print();      // за i = 1, ..., 5
 }
 p.print();
 q.print();
}
```

В резултат от изпълнението на main се получава:

```
1/2
destruct number:1/2
2/3
destruct number:2/3
3/4
destruct number:3/4
4/5
destruct number:4/5
5/6
destruct number:5/6
1/8
2/9
destruct number:2/9
destruct number:1/8
```

От изпълнението се вижда, че деструкторът на класа rat е извикан толкова пъти, колкото пъти са извършвани дейности по създаване на обекти на класа rat. Пътвите пет извиквания на деструктора са при завършване на изпълнението на блока на оператора for, а последните две – при завършване на изпълнението на функцията main.

Във втория случай създаването и разрушаването на обекти се управлява от програмиста. Създаването става с new, а разрушаването

чрез delete. Операциите new се включват в конструкторите, а операциите delete – в деструктора на класа.

Пример:

```
rat *p = new rat(3,7); // търси в хийпа 8В, свързва адреса
                       // им с p, извиква конструктора
                       // rat(9,13) и инициализира паметта
(*p).print();
delete p;            // извиква деструктора, след което
                     // разрушава обекта
...
```

В този случай деструкторът само регистрира присъствието си. Получаваме:

```
3/7
destruct number: 3/7
```

14.11 Инициализиране на обекти на класове

Езикът C++ позволява обектите на класове (както и обикновените променливи) да бъдат инициализирани при дефиницията си и при извикването на функцията new. При обикновените променливи инициализаторът задава стойност на променливата, а при обектите - осигурява аргументи на конструкторите. Инициализацията на обект на клас се извършва по следните начини:

```
<име_на-клас> <обект>(<инициализатор>);|
<име_на-клас> <обект> = <инициализатор>;
```

Възможни са:

а) инициализаторът не е обект на класа

В този случай се създава обекта, след това се намира стойността на израза-инициализатор и се подава на подходящия конструктор (ако има такъв).

Пример: Нека в класа rat конструкторът е с прототип:

```
rat(int = 0; int = 1);
```

Инициализацията

```
rat p = 7;
```

ще създаде обекта p и ще извика конструктора rat(7), с който ще инициализира p. Ако в класа rat не беше дефиниран конструктор с един аргумент, опитът за тази инициализация щеше да е неуспешен.

б) инициализаторът е обект на класа

В този случай съществуват някои особености. Ако съществува явно дефиниран конструктор за присвояване, той се използва. В противен случай се използва подразбиращия се конструктор за копиране.

Конструктор за присвояване явно не е дефиниран

Тогава се извиква подразбиращия се системен конструктор за копиране.

Пример:

```
rat p(1,9);
rat q = p;
```

Създава се нов обект q с член-данни абсолютни копия на съответните член-данни на p.

В този случай възникват проблеми ако някоя член-данна на обекта е указател към динамичната памет, тъй като член-променливата на новия обект, който е указател към динамичната памет, е със същата стойност като на стария (указва към същата памет). В този случай става поделяне на компонента на обектите.

Пример: Ще използваме класа product, като ще направим някои модификации в него:

```
class product
{private:
  char* name;
  double price;
  int score;
 public:
  ~product();
  product();
  void print() const;
  bool is_better_from(product const &) const;
  char* get_name() const;
  double get_price() const;
  int get_score() const;
};
product::~product()
{delete name;
 cout << "destruct data for: " << this << endl;
}
```

```
product::product()
{cout << "name: ";
 cin >> s;
 name = new char[strlen(s)+1];
 strcpy(name, s);
 cout << "price: ";
 cin >> price;
 cout << "score: ";
 cin >> score;
 cout << "new data: " << this << endl;
}
...
void main()
{product p;
 product q = p;
}
```

В този случай дефинираният конструктор по подразбиране product() се извиква веднъж – при инициализирането на p. Тъй като няма явно дефиниран конструктор за присвояване, генерираният от системата конструктор за копиране откопирва член-данните на обекта p в q като член-данната name е поделена. При завършване на блока – тяло на main, първо се разрушава обектът q. За него се извиква деструкторът. Поделената памет се освобождава, след което се разрушава и q. Забележете, q е разрушен, но е разрушена и част от p - поделената динамична памет. После започва процедурата по разрушаването и на обекта p. Извиква се деструкторът, който се опитва да освободи вече освободена памет. Това води до грешка, чийто последствия са непредвидими.

Конструктор за присвояване явно е дефиниран

Вече дефинирахме един глупав конструктор за присвояване за класа rat и правихме експерименти с него. Ще напомним, че конструкторът за присвояване е член-функция от вида:

```
<име_на_клас>(<име_на_клас> const&)
{<тяло>}
```

Ще дефинираме подходящ конструктор за присвояване в класа product и ще го извикаме за да реализираме коректно инициализацията от последния пример.

```
product::product(product const & p)
```

```
   {name = new char[strlen(p.name)+1];
    strcpy(name, p.name);
    price = p.price;
    score = p.score;
   }
```
и включваме прототипа му
```
   product(product const & p);
```
в public-частта на класа product.
Тогава функцията
```
   void main()
   {product p;
    product q = p;
   }
```
вече работи добре. Дефинирани са два обекта p – инициализиран чрез конструктора по подразбиране product() и q – чрез дефинирания явно конструктор за присвояване. Деструкторът е извикан два пъти при завършване изпълнението на блока и разрушава обектите q и p.

Дефинираният конструктор за присвояване решава проблемите, възникващи при инициализацията на обект на клас product. Използва се при предаване на обект по стойност, а също при връщане на обект като стойност на функция. Като цяло обаче той не решава проблемите на операцията за присвояване.

Пример: Ако променим main по следния начин:
```
   void main()
   {product p, q;
    q = p;
   }
```
отново възникват проблеми. Присвояването q = p; ще промени член-данните на q, но q вече има част в динамичната памет, която *първо трябва да бъде освободена*. Налага се да бъде дефинирана нова операторна функция за присвояване. Последното ще направим по следния начин:
```
   product& product::operator=(product const & p)
   {cout << "assignment!\n";
    if(this != &p)
    {delete name;
     name = new char[strlen(p.name)+1];
     strcpy(name, p.name);
```

```
     price = p.price;
     score = p.score;
    }
    return *this;
   }
```

като ще включим прототипа й

```
   product& operator=(product const &);
```

в public-частта на класа.

Забелязваме, че операторната функция за присвояване извършва аналогични действия на тези на конструктора за присвояване. Разликата е, че тя извършва тези действия върху съществуващ обект, а не върху токущо създаден. Това налага предварително да бъде освободена динамичната памет, отделена за обекта.

Дефинираната операторна функция има за формален параметър константен псевдоним от клас product. По този начин се избягва създаването на нов обект и извикването на конструктора за присвояване. Въпреки, че не е задължително, използването на константен псевдоним е препоръчително. Това позволява на компилатора да следи за евентуална промяна на обекта – фактически параметър. Освен, че променя обекта, указван от this, операторната функция от примера връща като резултат псевдонима му. Като следствие, конструкцията p = q може да се разглежда като израз (p = q връща p), а също да бъде лява страна на израз. Изразът p = q = r е допустим и е еквивалентен на q = r; p = q;

В този пример реализирахме като член-функции на класа product конструктор за присвояване, операторна функция за присвояване и деструктор. Тези три функции се наричат “голямата тройка” или “канонична форма на класа”. На пратика те са задължителни при класове, използващи динамичната памет. Това не е просто добра идея, това е закон.

14.12 Масиви от обекти

Създаването на масив от обекти става по два начина:

- чрез дефиниция
- чрез функциите за динамично управление на паметта.

При първия начин масивът от обекти се създава при срещането на дефиницията (във функция или блок) и се разрушава при завършване

на изпълнението на функцията или при излизане от блока. Дефиницията може да бъде поставена където и да е в тялото на функцията или блока, като пред и след нея може да има изпълними оператори.

Примери: Ще използваме класа rat

```
а) {...
     rat x[10];
   }
```

В този случай конструкторът rat() е извикан 10 пъти. Конструиран е масивът от обекти x:

```
x[0]  x[1]  ...   x[9]
0/1   0/1   ...   0/1
```

При завършване изпълнението на блока деструкторът ~rat() ще бъде извикан също 10 пъти за да разруши последователно x[9], x[8], ..., x[0].

```
б) {rat x[10] = {rat(1,2), rat(5), 8, rat(1,7)};
   }
```

В този случай конструкторът rat(int = 0, int = 1) е извикан 10 пъти. Конструиран е масивът от обекти x:

```
x[0]  x[1]  x[3]  x[4]  x[5]  x[6]  ...   x[9]
1/2   5/1   8/1   1/7   0/1   0/1   ...   0/1
```

При завършване изпълнението на блока деструкторът ~rat() ще бъде извикан също 10 пъти за да разруши последователно x[9], x[8], ... x[0].

При втория начин, създаването и разрушаването на масив от обекти се управлява от програмиста. Отново създаването става чрез new, а разрушаването – с delete, като операторите new се включват в конструкторите, а операторите delete – в деструктора на класа, от който са обектите на масива.

Примери:

```
а){rat *px = new rat[10];
   delete[] px;
  }
```

В резултат, в хийпа се заделя блок от 80 байта (ако е възможно) и адресът на този блок се записва в px. Тъй като има дефиниран конструктор по подразбиране, конструкторът се извиква и px[i] (i=0, 1, ..., 9) се инициализират с рационалното число 0/1. При масивите, реализирани в динамичната памет, инициализация в явен

вид не може да се зададе. Разрушаването на px става чрез delete[] px;. Преди да прекъсне връзката между px и динамичната памет, операторът delete[] извиква деструктора за всеки от обектите на масива.

```
б) {rat *px = new rat[10];
    delete px;
   }
```

В резултат, в хийпа се заделя блок от 80 байта (ако е възможно) и адресът на блока се записва в px. Тъй като в класа има конструктор по подразбиране, този конструктор се извиква и px[i] (i=0, 1, ..., 9) се инициализират с рационалното число 0/1. Операторът delete px; извиква деструктора само на обекта px[0] и прекъсва връзката на px с динамичната памет. Компилаторът съобщава за грешка. Причината е, че px е масив от обекти в динамичната памет, а деструкторът на класа rat е извикан само за px[0]. Ще отбележим отново, че ако px беше масив в динамичната памет, но не от обекти на клас, операторът delete px; щеше да работи нормално.

```
в) {int size;
    cin >> size;
    rat* px = new rat[size];
    delete[size] px;
   }
```

В този случай деструкторът на класа rat се извиква size пъти. Реализацията на Visual C++ 6.0 пренебрегва size от delete, но за някои други реализации това не е така.

14.13 Приятелски класове и функции

Често е необходимо съвместното използване на два класа. Обектно-ориентираното програмиране налага капсолирането на данните. Достъпът до private-компонентите на даден клас от функция извън класа е забранено. В редица случаи това е сериозно затруднение. Например, дефинирани са два класа, представящи вектор и матрица съответно. Функцията, която ще реализира произведението на вектор с матрица ще трябва да има достъп до членовете и на двата класа. Един начин за реализирането на това е да се направят член-данните и на двата класа public. Това ще доведе до загубване на предимствата на капсолирането. Друг начин е да се използват

public функции на достъп, осъществяващи достъп до стойностите на член-променливите. Това води до забавяне на изпълнението на програмата. Трети начин за решаване на този проблем е декларирането на функции или класове – приятели на класа. Приятелите на даден клас (функции или класове) имат достъп до всички негови компоненти, т.е. членовете на класа са винаги public за функциите приятели. Ако клас е деклариран като приятел, всички негови член-функции стават функции приятели.

Примери за функции и класове приятели ще дадем в следващите части на тази глава и книгата.

14.14 Оператори. Предефиниране на оператори

Езикът C++ има богат набор от оператори. В него са дадени също средства за предефиниране на оператори.

Всеки оператор се характеризира с:

- позиция на оператора спрямо аргументите му;
- приоритет;
- асоциативност.

Позицията на оператора спрямо аргументите му го определя като: префиксен (операторът е пред единствения му аргумент), инфиксен (операторът е между аргументите си) и постфиксен (операторът е след аргумента си).

Пример: Операторът / е инфиксен (4/8), операторът + е както инфиксен, така и префиксен (2+8, +78), а операторът ++ е както постфиксен, така и префиксен.

Приоритетът определя реда на изпълнение на операторите в операторен терм. Оператор с по-висок приоритет се изпълнява преди оператор с по-нисък приоритет.

Пример: Приоритетът на * и / е по-висок от този на + и -.

Асоциативността определя реда на изпълнение на оператори с еднакъв приоритет в операторен терм. В C++ има лявоасоциативни и дясноасоциативни оператори. Лявоасоциативните оператори се изпълняват отляво надясно, а дясноасоциативните – отдясно наляво.

В C++ не могат да се дефинират нови оператори, но всеки съществуващ едноаргументен или двуаргументен оператор с изключение на ::, ?:, ., *, # и ## може да бъде предефиниран от програмиста, стига поне един операнд на оператора да е обект на някакъв клас.

Например, възможно е да се предефинират операторите +, -, * и /, така че да могат да събират, изваждат, умножават и делят рационални числа. Тогава вместо sum(p, q), sub(p, q), mult(p, q) и quot(p, q) ще можем да пишем p+q, p-q, p*q и p/q, което безспорно е много по-удобно.

Предефинирането се осъществява чрез дефиниране на специален вид функции, наречени **операторни функции**. Последните имат синтаксис като на обикновените функции, но името им се състои от запазената дума operator, следвана от мнемоничното означение на предефинирания оператор. Когато предефинирането на оператор изисква достъп до компонентите на класове, обявени като private или protected, операторната дефиниция трябва да е член-функция или функция-приятел на тези класове. Предефинираният оператор запазва всички характеристики на оригиналния.

Предефинирането може да стане по два начина:

- чрез функция-приятел;
- чрез член-функция.

Чрез примери ще покажем тези два начина.

Предефиниране чрез функция-приятел

Задача 126. Да се предефинират операторите +, -, * и / така, че да могат да бъдат използвани за събиране, изваждане, умножение и деление на рационални числа.

Програма Zad126_1.cpp решава задачата. В public частта на класа rat са включени декларациите на предефинираните оператори, предшествани от запазената дума friend:

```
friend rat operator+(rat const &, rat const &);
friend rat operator-(rat const &, rat const &);
friend rat operator*(rat const &, rat const &);
friend rat operator/(rat const &, rat const &);
```

а след дефиницията на функцията main са дадени и техните дефиниции.

```
// Program Zad126_1.cpp
#include <iostream.h>
class rat
{private:
```

```
  int numer;
  int denom;
 public:
  rat(int=0, int=1);
  ~rat()
  {cout << "destruct number: " << numer <<
         "/" << denom << endl;
  }
  void read();
  int get_numer() const;
  int get_denom() const;
  void print() const;
  friend rat operator+(rat const &, rat const &);
  friend rat operator-(rat const &, rat const &);
  friend rat operator*(rat const &, rat const &);
  friend rat operator/(rat const &, rat const &);
};
rat::rat(int a, int b)
{numer = a;
 denom = b;
 cout << "construct! \n";
}
void rat::read()
{cout << "numer: ";
 cin >> numer;
 do
 {cout << "denom: ";
  cin >> denom;
 } while (denom == 0);
}
int rat::get_numer() const
...
int rat::get_denom() const
...
void rat::print() const
...
int main()
{rat p(1,3), q(2,5), r(p+q);
```

```
 r.print();
 r = p-q-q;
 r.print();
 return 0;
}
// предефиниране на оператора +
rat operator+(rat const & r1, rat const & r2)
{rat r(r1.numer*r2.denom +
       r2.numer*r1.denom,
       r1.denom*r2.denom);
 return r;
}
// предефиниране на оператора -
rat operator-(rat const & r1, rat const & r2)
{rat r(r1.numer*r2.denom-
       r2.numer*r1.denom,
       r1.denom*r2.denom);
 return r;
}
// предефиниране на оператора *
rat operator*(rat const & r1, rat const & r2)
{rat r(r1.numer*r2.numer,
       r1.denom*r2.denom);
 return r;
}
// предефиниране на оператора /
rat operator/(rat const & r1, rat const & r2)
{rat r(r1.numer*r2.denom,
       r1.denom*r2.numer);
 return r;
}
```

Забележки:

1) Изразът p+q се интерпретира като извикване на операторната функция operator+(p, q).
2) Запазва се асоциативността. Изразът p-q-r се интерпретира като (p-q)-r.

Предефиниране чрез член-функция

В този случай първият аргумент на член-функцията трябва да е обект на класа и при дефинирането на операторната функция не се задава като параметър. Ако това не е така, операцията не може да се предефинира като член-функция.

Ще модифицираме предишната програма като дефинираме операторните функции за събиране, изваждане, умножение и деление като член-функции на класа rat.

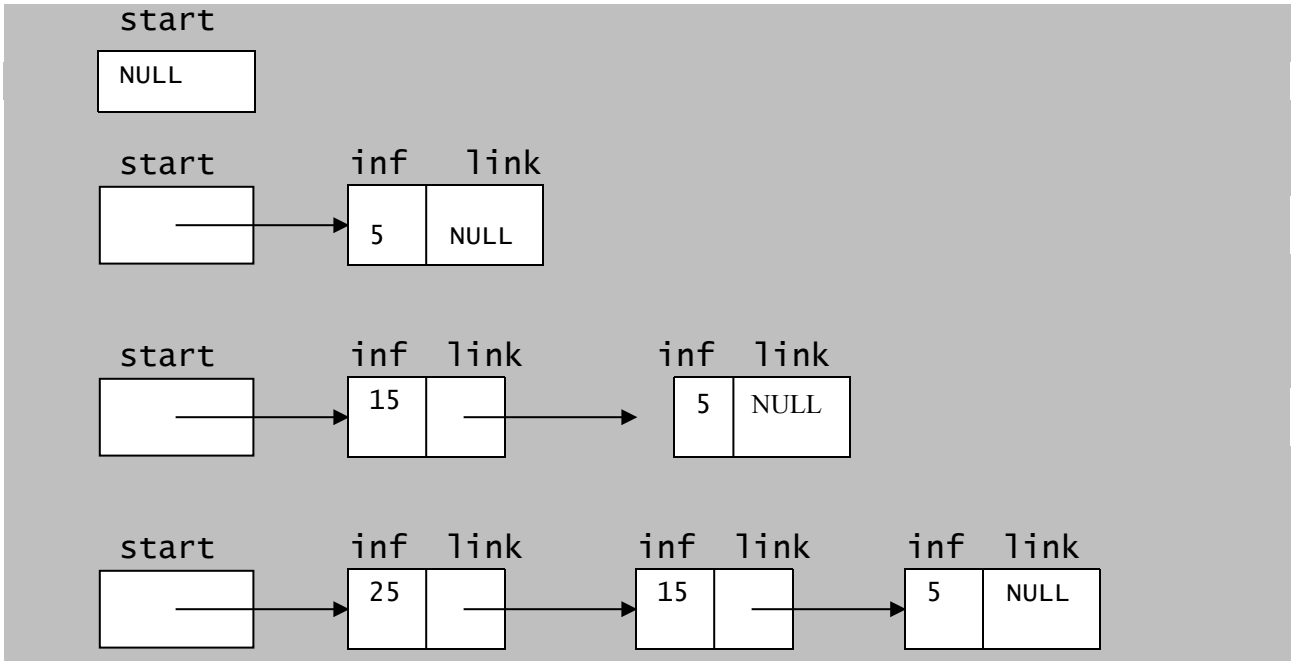
```
// Program Zad126_2.cpp
#include <iostream.h>
class rat
{private:
  int numer;
  int denom;
 public:
  rat(int=0, int=1);
  ~rat()
  {cout << "destruct number: " << numer
      << "/" << denom << endl;
  }
  void read();
  int get_numer() const;
  int get_denom() const;
  void print() const;
  rat operator+(rat const &) const;
  rat operator-(rat const &) const;
  rat operator*(rat const &) const;
  rat operator/(rat const &) const;

};
rat::rat(int a, int b)
...
void rat::read()
...
int rat::get_numer() const
...
int rat::get_denom() const
```

```
...
void rat::print() const
...
int main()
{rat p(1,3), q(2,5), r(p+q);
 r.print();
 r = p-q-q;
 r.print();
 return 0;
}
rat rat::operator+(rat const & r1) const
{rat r(numer*r1.denom + r1.numer*denom,
       denom*r1.denom);
 return r;
}
rat rat::operator-(rat const & r1) const
{rat r(numer*r1.denom - denom*r1.numer,
       denom*r1.denom);
 return r;
}
rat rat::operator*(rat const & r1) const
{rat r(numer*r1.numer,
       denom*r1.denom);
 return r;
}
rat rat::operator/(rat const & r1) const
{rat r(numer*r1.denom,
       denom*r1.numer);
 return r;
}
```

Ще отбележим, че в този случай изразът p+q се интерпретира като p.operator+(q).

14.15 Приложение на средствата за работа с динамичната памет

Ще конструираме клас stack, който ще реализира свързаното представяне на стек от цели числа. Фиг. 14.10 илюстрира това представяне.



Фиг. 14.10 Свързано представяне на стек

Забелязваме, че има указател start, който в първия случай представя празен стек, а в останалите случай – непразен, като сочи двойна кутия с информационна част (inf) от тип int и свързваща част (link) от типа на start. Това представяне ще реализираме по следния начин:

```
struct elem
{int inf;
 elem* link;
} *start = NULL, *p;
```

След тази дефиниция start представя празен стек. Включването на елемента 5 можем да направим чрез изпълнение на следните действия:

```
p = start;
start = new elem;
start->inf = 5;
start->link = p;
```

Включването на 15 ще направим по аналогичен начин

```
p = start;
start = new elem;
start->inf = 15;
start->link = p;
```

а на 25 – чрез

```
p = start;
start = new elem;
start->inf = 25;
start->link = p;
```

Тези разсъждения показват, че който и да е елемент x може да се **включи** в стека чрез изпълнение на фрагмента:

```
p = start;
start = new elem;
start->inf = x;
start->link = p;
```

Изключването на елемент от последния стек от фиг. 14.10 води до получаване на стека, илюстриран на по-горната стъпка на същата фигура и може да се реализира така:

```
p = start;
x = start->inf;
start = start->link;
delete p;
```

В x е запомнен изключеният елемент. Така получаваме следната непълна реализиция на свързаното представяне на стек от цели числа:

```
struct elem
{int inf;
 elem* link;
};
class stack
{public:
  stack(); // конструктор по подразбиране
    void push(int const&); // член-функция за включване на елемент
  int pop(int & x);  // член-функция за изключване на елемент
 private:
  elem *start;
};
stack::stack()
{start = NULL;
}
void stack::push(int const& s)
```

```
{elem* p = start;
 start = new elem;
 start->inf = s;
 start->link = p;
}
int stack::pop(int & s)
{elem *p;
 if (start)
 {s = start->inf;
  p = start;
  start = start->link;
  delete p;
  return 1;
 }
return 0;
}
```

Забелязваме, че pop връща 1 ако операцията изключване е възможна и 0 – ако не е.

Тъй като обектите на класа stack са реализирани в динамичната памет, за него трябва да реализираме каноничното представяне.

деструктор

Единствената член-данна на класа stack е указател към динамичната памет, където е разположен стекът. Затова деструкторът трябва да изтрие стека от паметта. Тъй като изтриването на стек ще е необходимо и за други цели, ще го реализираме чрез член-функцията delstack() на класа.

```
void stack::delstack()
{elem *p;
 while (start)
 {p = start;
  start = start->link;
  delete p;
 }
}
```

Тогава деструкторът ~stack() има вида:

```
stack::~stack()
{delstack();
}
```

конструктор за присвояване

Конструкторът за присвояване

```
stack(stack const & r);
```

откопирва стека r в неявния параметър и ще го реализираме по следния начин:

```
stack::stack(stack const & r)
{copy(r);
}
```

където copy(r) ще дефинираме като член-функция на stack и тя ще реализира копиране на стека r в неявния параметър.

операторна функция за присвояване

Ще я реализираме така:

```
stack& stack::operator=(stack const& r)
{if (this != &r)
 {delstack();
  copy(r);
 }
 return *this;
}
```

Копирането ще реализираме чрез член-функцията copy(stack const &r). За целта ще сканираме елементите на r (без да ги разрушаваме) и ще ги запишем на друго място в динамичната памет.

```
void stack::copy(stack const & r)
{if (r.start) // r не е празен
{elem *p = r.start, *q = NULL, *s=NULL;
 start = new elem;
 start->inf = p->inf;
 start->link = q;
 q = start;
 p = p->link;
 while (p)
 {s = new elem;
  s->inf = p->inf;
  q->link = s;
  q = s;
  p = p->link;
 }
 q->link = NULL;
```

```
   }
   else start = NULL;
  }
```

Класът stack, реализиращ свързаното представяне на стек, има вида:

```
   struct elem
   {int inf;
    elem* link;
   };
   class stack
   {public:
     stack();
     ~stack();
     stack(stack const &);
     stack& operator=(stack const & r);
     void push(int const&);
     int  pop(int & x);
     bool empty() const;
     void print();
    private:
     elem *start;
     void delstack();
     void copy(stack const&);
   };
   stack::stack()
   {start = NULL;
   }
   stack::~stack()
   {delstack();
   }
   stack::stack(stack const & r)
   {copy(r);
   }
   stack& stack::operator=(stack const& r)
   {if (this != &r)
   {delstack();
    copy(r);
   }
   return *this;
```

```
}
void stack::delstack()
{elem *p;
 while (start)
 {p = start;
  start = start->link;
  delete p;
 }
}
void stack::copy(stack const & r)
{if(r.start)
{elem *p = r.start, *q = NULL, *s=NULL;
 start = new elem;
 start->inf = p->inf;
 start->link = q;
 q = start;
 p = p->link;
 while (p)
 {s = new elem;
  s->inf = p->inf;
  q->link = s;
  q = s;
  p = p->link;
 }
 q->link = NULL;
}
else start = NULL;
}
void stack::push(int const& s)
{elem* p = start;
 start = new elem;
 start->inf = s;
 start->link = p;
}
int stack::pop(int & s)
{if (start)
 {s = start->inf;
  elem *p= start;
```

```
     start = start->link;
     delete p;
     return 1;
    }
    else return 0;
   }
   void stack::print()
   {int x;
    while (pop(x))
      cout << x << " ";
    cout << endl;
   }
   bool stack::empty() const
   {return start==NULL;
   }
```

В него са добавени и член-функциите:

```
   void print();
```

която извежда елементите на стек и

```
   bool empty() const;
```

проверяваща дали стек е празен.

Следващите програми използват този клас.

Задача 127. Да се напише функция void sortstack(stack s, stack & ns), която сортира елементите на стека от цели числа s по метода на пряката селекция. Стекът ns съдържа резултата.

Функцията

```
   void minstack(stack s, int& min, stack &newst);
```

намира минималния елемент на стека s, а също стека newst, съдържащ елементите на s без минималния.

```
   void minstack(stack s, int& min, stack &newst)
   {int x;
    s.pop(min);
    while (s.pop(x))
    if (x<min)
     {newst.push(min);
      min = x;
     }
```

```
 else newst.push(x);
 }

void sortstack(stack s, stack& ns)
{int min;
 while (!s.empty())
 {stack s1;
  minstack(s, min, s1);
  ns.push(min);
  s = s1;
 }
}
```

Едно приложение на структурата от данни стек е пресмятането на стойности на изрази.

Задача 128. От клавиатурата е въведена записана без грешка формула от вида:

```
<формула> ::=  <цифра>|
               M(<формула>, <формула>)|
               m(<формула>, <формула>)
<цифра> ::= 0|1|...|9,
```

където M означава функция за намиране на максимума на две цифри, а m – функция за намиране на минимума на две цифри. Въвеждането продължава до срещане на символа ‘.’. Например, стойността на формулата 8 е 8, а стойността на M(1,m(9,6)) е 6.

Функцията formula решава задачата. Тя използва помощен стек от символите ‘M’, ‘m’ и цифрите и реализира следния алгоритъм. Ако прочетеният символ е ‘M’, ‘m’, ‘0’, ...,’9’, се записва в стека s. Ако е ‘)’, от s се изключват три елемента. Тъй като въвежданата формула е правилна (по условие) изключените елементи са два операнда и операция (M - max или m - min). Извършва се операцията и полученият резултат се включва в стека. Останалите символи (интервали, ‘(‘, ‘,’, символите за преминаване на нов ред) се пропускат.

Помощният стек от символи ще реализираме като обект на *клас stack от символи*, който получаваме като заместим в дефиницията на класа stack типа int с char.

```
int formula()
{char c, x, y, op;
 stack st;
 cin >> c;
 while (c!='.')
 {if (c=='m'||c=='M'||c>='0'&&c<='9') st.push(c);
  else
     if (c==')')
     {st.pop(y);
      st.pop(x);
      st.pop(op);
      switch (op)
      {case 'm': if(x<y)c=x;else c=y; break;
       case 'M': if(x>y)c=x;else c=y;
      }
      st.push(c);
     }
     cin >> c;
 }
 st.pop(c);
 return (int)c-(int)'0';
}
```

14.16 Шаблони на функции и класове

В предните разглеждания създадохме класа stack, реализиращ стек от цели числа. След това за други цели се наложи да променим този клас в клас, реализиращ стек от символи. Промяната беше елементарна – просто заменихме типа int на елементите на стека с char. Възможно е обаче в рамките на една и съща програма да е необходимо конструирането на няколко стека от различен тип данни. Така възниква необходимостта от средства, реализиращи *класове, зависещи от параметри*, задаващи типове данни и при конкретни приложения параметрите да се конкретизират.

Такива средства са **шаблоните**. Те позволяват създаването на класове, използващи неопределени (хипотетични) типове данни за своите аргументи и по такъв начин позволяват да бъдат описвани “обобщени” типове данни. Използват се за изграждане на общоцелеви

класове-контейнери (стекове, опашки, списъци и др.). Например, чрез шаблон може да се дефинира обобщен клас за стек с неопределен тип на елементите, след което от шаблона да се получат специфични класове (клас, реализиращ стек от реални числа; клас, реализиращ стек от символи и т.н.). При наличие на шаблони на класове възниква необходимостта и от шаблони на функции, използващи шаблони на класове. Например, искаме на реализираме сортиране на елементите на шаблон на стек. Налага се да дефинираме шаблони на функциите за намиране на минимален елемент на стек с произволен тип на елементите и сортиране на елементите на стек с произволен тип на елементите.

14.16.1 Шаблони на функции

Дефиницията на шаблон на функция е дадена на Фиг. 14.11.

Дефиниция на шаблон на функция

<дефиниция_на_шаблон_на_функция> ::=
template < class <параметър> {,class <параметър>}$_{опц}$ >
<тип_на_функция> <име_на_функция>(<формални_параметри>)
<тяло>

където

- <параметър> и <име_на_функция> са идентификатори;
- <формални_параметри> и <тяло> се определят както при дефиниция на функция. В тях са използвани указаните параметри на шаблона, вместо конкретните типове.

Фиг. 14.11 Дефиниция на шаблон на функция

Дефиницията започва със запазената дума template (шаблон), следвана от списък от параметри на шаблона, които трябва да участват като типове на аргументи на дефинираната като шаблон функция.

Използването на дефинираните шаблони на функции става чрез обръщение към “обобщената” функция, която шаблонът дефинира, с параметри от конкретен тип. Компилаторът генерира т. нар. **шаблонна функция**, като замества параметрите на шаблона с типовете на

съответните фактически параметри. *При това заместване не се извършват преобразувания на типове.*

Задача 129. Да се напише програма, която дефинира шаблон на процедура за въвеждане елементите на масив и шаблон на функция, намираща минималния елемент на масив от елементи, които могат да се сравняват.

```
// Procedure Zad129.cpp
#include <iostream.h>
template <class T>
void read(int n, T* a)
{for (int i = 0; i<=n-1; i++)
 {cout << "a[" << i << "]= ";
  cin >> a[i];
 }
}
template <class T>
T minarray(int n, T* a)
{T min = a[0];
 for (int i = 1; i<=n-1; i++)
  if (a[i]<min) min = a[i];
 return min;
}
int main()
{cout << "n: ";
 int n;
 cin >> n;
 int a[10];
 read(n, a);
 cout << minarray(n, a) << endl;
 double b[10];
 read(n, b);
 cout << minarray(n, b) << endl;
 return 0;
}
```

14.16.2 Шаблони на класове

Дефинирането на шаблон на клас се състои от декларация на шаблона и дефиниране на член-функциите му. На Фиг. 14.12 е даден непълно синтаксисът на декларацията на шаблон на клас.

Декларация на шаблон на клас

<декларация_на_шаблон_на_клас> ::=

template < class <параметър>[=<име_на_тип>]$_{опц}$

{,class <параметър>[=<име_на_тип>]$_{опц}$}$_{опц}$

>

class <име_на_шаблон_на_клас>

<тяло>;

където

- <параметър>, <име_на_шаблон_на_клас> и <име_на_тип> са идентификатори. В <тяло> са използвани указаните параметри на шаблона, вместо конкретните типове.

Фиг. 14.12 Декларация на шаблон на клас

Броят на параметрите на шаблон на клас е произволен. Параметрите могат да участват на произволни места в дефиницията на шаблона. Освен това е възможно някои или всички параметри на шаблона да са подразбиращи се. Това се осъществява чрез добавяне на инициализацията =<име_на_тип> след името на параметъра. В този случай, при пропускане на параметър, се използва подразбиращият се тип.

Пример: Декларацията

```
template <class T, class S = int> class CLASS
{public:
 T func1(T x, S y);
 S func2(T x, S y);
 private:
  T a;
  S b;
};
```

определя шаблон на клас CLASS с два параметъра T и S, като вторият е подразбиращ се – при неуказване, S се интерпретира като тип int.

Дефинирането на член-функции на шаблон се осъществява по два начина – като вградени и не като вградени (описани извън декларацията). При дефинирането на вградените член-функции няма особености. Ако е необходимо, използват се параметрите на класа.

Пример:

```
template <class T, class S = int> class CLASS
{public:
  T func1(T x, S y)     // вградена член-функция
  {cout << "func1 \n";
   return x;
  }
  S func2(T x, S y);
 private:
  T a;
  S b;
 };
```

В другия случай дефиницията се предшества от

template <списък_от_параметри>

а пълното име на член-функцията на шаблона се получава с префикса

<име_на_шаблон_на_клас> **<** <параметър> {,<параметър>}$_{опц}$ **>**

Пример:

```
template <class T, class S>
S CLASS<T, S>::func2(T x, S y)
{cout << "func2 \n";
 return y;
}
```

Забележка: Префиксът се използва и когато член-функция на шаблона е от тип шаблон на клас, указател или псевдоним на шаблон на клас.

Шаблоните на класове не са истински класове, а описания, които се използват от компилатора за създаване на конкретни (шаблонни) класове. Наричат се още **специализации на шаблона на класа**.

Фиг. 14.13 дава дефиницията на шаблонни класове.

Дефиниция на шаблонен клас
<дефиниция_на_шаблонен_клас> ::=

```
        <име_на_шаблон_на_клас> < <тип>, <тип>,... >
  <тип> ::= <име_на_тип>
```

Фиг. 14.13 Дефиниция на шаблонен клас

Ако някой <тип> е пропуснат, използва се подразбиращият се, ако декларацията на шаблона е с подразбиращи се параметри, или се съобщава за грешка. При срещане на декларация на шаблонен клас, на базата на зададените типове, компилаторът генерира съответен шаблонен клас.

Пример:

а) typedef CLASS<int, double> obj1;

дефинира класа obj1, който е специализация на шаблона на класа CLASS при T – int и S – double. Дефиницията

obj1 o1;

определя o1 за обект на класа obj1, който може да се обръща към public-членовете на obj1, т.е. допустимо е обръщението

o1.func1(5, 10.87);

б) typedef CLASS<int> obj2;

дефинира класа obj2, който е специализация на шаблона на класа CLASS при T – int и S – int. Дефиницията

obj2 o2;

определя o2 за обект на класа obj2, който може да се обръща към public-членовете на obj2, т.е. допустимо е обръщението

o2.func2(5,8);

Забележка: Ако на CLASS и двата параметър бяха подразбиращи се, щеше да е възможна и специализацията:

typedef CLASS <> obj3; // ъгловите скоби <> трябва да присъстват

Като илюстрация на казаното ще дефинираме и използваме шаблон на класа stack.

Задача 130. Да се дефинират шаблон на класа stack и шаблонна функция за сортиране на елементите на шаблон на стек.

Програма Zad130.cpp решава задачата.

// Program Zad130.cpp

```
#include <iostream.h>
template <class T>
struct elem
{T inf;
 elem* link;
};
template <class T>
class stack
{public:
  stack();
  ~stack();
  stack(stack const &);
  stack& operator=(stack const & r);
  void push(T const&);
  int  pop(T & x);
  bool empty() const;
  void print();
 private:
  elem<T> *start; // указател към шаблона на структурата elem
  void delstack();
  void copy(stack const&);
};
template <class T>
stack<T>::stack()
{start = NULL;
}
template <class T>
stack<T>::~stack()
{delstack();
}
template <class T>
stack<T>::stack(stack<T> const & r)
{copy(r);
}
template <class T>
stack<T>& stack<T>::operator=(stack<T> const& r)
{if (this != &r)
{delstack();
```

```
 copy(r);
}
return *this;
}
template <class T>
void stack<T>::delstack()
{elem<T> *p;
 while (start)
 {p = start;
  start = start->link;
  delete p;
 }
}
template <class T>
void stack<T>::copy(stack<T> const & r)
{if (r.start)
{elem<T> *p = r.start, *q = NULL, *s=NULL;
 start = new elem<T>;
 start->inf = p->inf;
 start->link = q;
 q = start;
 p = p->link;
 while (p)
 {s = new elem<T>;
  s->inf = p->inf;
  q->link = s;
  q = s;
  p = p->link;
 }
 q->link = NULL;
}
else start = NULL;
}
template <class T>
void stack<T>::push(T const& s)
{elem<T> *p = start;
 start = new elem<T>;
 start->inf = s;
```

```
 start->link = p;
}
template <class T>
int stack<T>::pop(T & s)
{if (start)
 {s = start->inf;
  elem<T> *p= start;
  start = start->link;
  delete p;
  return 1;
 }
 else return 0;
}
template <class T>
void stack<T>::print()
{T x;
 while (pop(x))
   cout << x << " ";
 cout << endl;
}
template <class T>
bool stack<T>::empty() const
{return start == NULL;
}
template <class T>
void minstack(stack<T> s, T& min, stack<T> &newst)
{T x;
 s.pop(min);
 while (s.pop(x))
 if (x<min)
  {newst.push(min);
   min = x;
  }
 else newst.push(x);
}
template <class T>
void sortstack(stack<T> s, stack<T>& ns)
{T min;
```

```
    while (!s.empty())
      {stack<T> s1;
       minstack(s, min, s1);
       ns.push(min);
       s = s1;
      }
   }
   void main()
   {typedef stack<int> IntStack;
    IntStack s, s1;
    s.push(10); s.push(1); s.push(5); s.push(12);
    s.push(8); s.push(14); s.push(19); s.push(0);
    sortstack(s, s1);
    s1.print();
   }
```

Във функцията main чрез

```
   typedef stack<int> IntStack;
```

е дефиниран класът IntStack. Освен това са дефинирани два шаблона на функции - за намиране на минимален елемент на стек с елементи от тип T и за сортиране на елементи на стек от тип T.

Шаблонът на клас дефинира съвкупност от класове. Понякога е по-удобно за някакъв тип данни член-функция на шаблона на класа да се реализира по различен алгоритъм. В C++ е възможно предефинирането на член-функция на шаблон на клас за конкретен тип. Този процес се нарича **специализация на член-функция**.

Пример: Член-функцията print() на шаблона на stack извежда елементите на стек от тип T. Ако стекът, за който я използваме е от символи – ще бъдат изведени символите. Ако е необходимо само в този случай да се извеждат ASCII-кодовете на символите, може да бъде предефинирана член-функцията print() като самостоятелна функция от вида:

```
   void stack<char>::print()
   {char x;
    while (pop(x))
      cout << (int)x << " ";
    cout << endl;
   }
```

Последната ще бъде използвана само при обръщение за извеждане на стек от символи, т.е.

```
typedef stack<char> CharStack;   // дефиниция на клас CharStack
CharStack s;         // s е обект на класа CharStack
s.push('u'); s.push('b');
s.push('p'); s.push('x');
s.print();  // ще се изведат ASCII-кодовете
            // на символите на стека.
```

Шаблоните на класове могат да имат за *приятели* класове, функции, шаблони на класове и шаблони на функции.

Пример: Нека имаме дефинициите:

```
class CLASS1 {...};
double func1(...){...}
template <class T> class CLASS2{...};
template <class T> double func2(T){..};
```

В дефиницията на шаблон на нов клас може да се въведат декларациите:

```
friend class CLASS1;
friend double func1(...);
friend class CLASS2<int>;  // само специализацията за int на
                           // шаблона на класа CLASS2 е приятел
friend double func2(double);  // само специализацията за
                              // double е приятел
friend class CLASS2<T>;       // всеки обект на шаблона
                              // на класа CLASS2 е приятел
friend double func2(T);       // всяка функция, създадена
                              // от шаблона, е приятел.
```

Допълнителна литература

1. B. Stroustrup, C++ Programming Language. Third Edition, Addison – Wesley, 1997.
2. К. Хорстман, Принципи на програмирането със C++, София, ИК СОФТЕХ, 2000.
3. Ал. Стивънс, Кл. Уолнъм, C++ БИБЛИЯ, София, АЛЕКССОФТ, 2000.
4. Ст. Липман, Езикът C++ в примери, “КОЛХИДА ТРЕЙД” КООП, София, 1993.