11

Структури

11.1 Структура от данни запис

Логическо описание

Записът е съставна статична структура от данни, която се определя като крайна редица от фиксиран брой елементи, които могат да са от различни типове. Достъпът до всеки елемент от редицата е пряк и се осъществява чрез име, наречено **поле на записа**.

Физическо представяне

Полетата на записа се представят последователно в паметта.

Примери:

1. Данните за студент от една група (име, адрес, факултетен номер, оценки по изучаваните предмети) могат да се зададат чрез запис с четири полета.

2. Данните за книга от библиотека (заглавие, автор, година на издаване, издателство, цена) могат да се зададат чрез запис с пет полета.

3. Комплексно число може да се зададе чрез запис с две реални полета.

В езика C++ записите се реализират чрез структури. Ще разгледаме последните в развитие. Отначало ще опишем възможностите им на ниво -език C.

11.2 Дефиниране и използване на структури

Една структура се определя чрез имената и типовете на съставящите я полета. Фиг. 11.1 дава непълна дефиниция на структура.

```
Дефиниция на структура
<дефиниция_на_структура> ::= struct <име_на_структура>
            {<дефиниция_на_полета>;
             {<дефиниция_на_полета>;}опц
            };
<име_на_структура> ::= <идентификатор>
<дефиниция_на_полета> ::= <тип> <име_на_поле>{, <име_на_поле>}опц
<тип> ::= <име_на_тип>|<дефиниция_на_тип>
<име_на_поле> ::= <идентификатор>
```

Фиг. 11.1 Дефиниция на структура

Структурите, дефинирани по този начин, могат да се използват като типове данни. Имената на полетата в рамките на една дефиниция на структура трябва да са *различни* идентификатори.

Примери:

1.
```
struct complex
   {double re, im;};
```

задава структура с име complex с две полета с имена re и im от тип double. Чрез нея се задават комплексните числа.

2.
```
struct book
   {char name[41], author[31];
    int year;
    double price;
   };
```

задава структура с име book с четири полета: *първо поле* с име name от тип символен низ с максимална дължина 40 и определящо името на книгата; *второ поле* с име author от тип символен низ с максимална дължина 30, определящо името на автора на книгата; *трето поле* с име year от тип int, определящо годината на издаване и *четвърто поле* с

име price от тип double и определящо цената на книгата. Чрез тази структура се задава информация за книга.

3. struct student
   {int facnum;
    char name[36];
    double marks[30];
   };

задава структура с име student и с три полета: *първо поле* с име facnum от тип int, означаващо факултетния номер на студента; *второ поле* с име name от тип символен низ с максимална дължина 35, определящо името на студента и *трето поле* с име marks от тип реален масив с 30 компоненти и означаващо оценките от положените изпити.

Възможно е за име на структура, на нейно поле и на произволна променлива на програмата да се използва един и същ идентификатор. Но тъй като това намалява читаемостта на програмата, засега не препоръчваме използването му.

Допуска се влагане на структури, т.е. поле на структура може да е от тип структура.

Пример: Допустими са дефинициите:

```
struct xx
{int a, b, c;
};
struct pom
{int a;
 double b;
 char c;
 xx d;
};
```

Не е възможно обаче поле на структура да е от тип, съвпадащ с името на структурата.

Пример: Не е допустима дефиниция от вида

```
struct xxx{
   xxx member;    // опит за рекурсивна дефиниция
};
```

тъй като компилаторът не може да определи размера на xxx. Допустима е обаче дефиницията:

```
struct xxx{
   xxx* member;
};
```

Допълнение: За да е възможно две дефиниции на структури да се обръщат една към друга, е необходимо пред дефинициите им да се постави декларацията на втората структура. Например, ако искаме дефиницията на структурата list да използва дефиницията на структурата link и обратно, ще трябва да ги подредим по следния начин:

```
struct list;      // декларация на втората структура
struct link{
   link* pred;
   link* succ;
   list* member;
};
struct list{
   link* head;
};
```

Дефиницията на структура не предизвиква отделянето на памет за съхраняване на компонентите ѝ. Може да се постави извън функция, в началото на функция или в началото на блок. Местоположението на дефиницията определя областта на името на структурата – съответно за всички функции след дефиницията, в рамките на функцията и в рамките на блока. Най-често дефиницията се задава пред първата функция на програмата и така става достъпна за всички функции.

Тъй като дефинирането на структура чрез задаване на името на структурата определя нов тип данни, ще определим множеството от стойности и операциите и вградените функции, свързани с него.

Множество от стойности

Множеството от стойностите на една структура се състои от всички крайни редици от по толкова елемента, колкото са полетата ѝ, като всеки елемент е от тип, съвместим с типа на съответното поле.

Примери:

1. Множеството от стойности на структурата complex се състои от всички двойки от реални числа.

2. Множеството от стойности на структурата book се състои от всички четворки от вида:

{char[41], char[31], int, double}.

3. Множеството от стойности на структурата student се състои от всички тройки от вида:

{int, char[36], double[30]}.

Променлива величина, множеството от допустимите стойности, на която съвпада с множеството от стойности на дадена структура, е променлива от дадения тип структура. Променлива от тип структура се дефинира в областта на структурата по следния начин (Фиг. 11.2):

Дефиниция на променлива от тип структура

<дефиниция_на_променлива_от_тип_структура> ::=
[struct]$_{опц}$ <име_на_структура>
<променлива> [={<редица_от_изрази>}]$_{опц}$
{,<променлива> [={<редица_от_изрази>}]$_{опц}$}$_{опц}$;|
struct {<дефиниция_на_полета>;
{<дефиниция_на_полета>;}$_{опц}$
}<променлива> [={<редица_от_изрази>}]$_{опц}$
{,<променлива> [={<редица_от_изрази>}]$_{опц}$}$_{цоп}$;
<променлива> ::= <идентификатор>
<редеица_от_изрази> ::= <израз>|
<израз>,<редица_от_изрази>

Фиг. 11.2 Дефиниция на променлива от тип структура

В C++ използването на запазената дума struct не е задължително. Някои програмисти го използват заради яснотата на кода. Конструкцията {<редица_от_изрази>} предизвиква инициализация на дефинирана променлива. Изразите, изредени във фигурните скоби, се разделят със запетая. Всеки израз инициализира поле на структурата и трябва да е от тип, съвместим с типа на съответното поле. Ако са записани по-

малко изрази от броя на полетата, компилаторът допълва останалите с нулевите стойности за съответния тип на поле.

Примери:

```
complex z1, z2 = {5.6, -8.3}, z3;
book b1, b2, b3;
struct student s1 = {44505, “Ivan Ivanov”, {5.5, 6, 5, 6}};
struct
{int x;
 double y;
} p, q = {-2, -1.6};
pom y;
```

Последната дефиниция от примера по-горе дефинира променливите p и q като променливи от тип структурата

```
struct
{int x;
 double y;
}
```

която участва с дефиницията, а не с името си.

Достъпът до полетата на структура е пряк. Един начин за неговото осъществяване е чрез променлива от тип структурата, като променливата и името на полето на структурата се разделят с оператора точка (Фиг. 11.3). Получените конструкции са *променливи* от типа на полето и се наричат **полета на променливата** от тип структура или **член-данни на структурата**, свързани с променливата.

Операторът . е лявоасоциативен и има приоритет равен на този на () и [].

Поле на структура

<поле_на_променлива_структура> ::=
    <променлива_структура>.<име_на_поле>

Фиг. 11.3 Поле на структура

Примери:

С променливите z1, z2, z3 се свързват променливите от тип double: z1.re, z1.im, z2.re, z2.im, z3.re и z3.im

a с b1, s1 и y –

b1.name - от тип char [41], b1.author – от тип char [31],
b1.year - от тип int, b1.price - от тип double,
s1.facnum – от тип int, s1.name - от тип char [36],
s1.marks - от тип double [30], y.a - от тип int,
y.b - от тип double, y.c - от тип char,
y.d - от тип xx и
с полета y.d.a, y.d.b и y.d.c от тип int.

Дефинирането на променлива от тип структура свързва променливата с множеството от стойности на съответната структура. След дефинициите от примера по-горе, променливите z1, z2 и z3 се свързват с множеството от стойностите на типа complex. При това свързване z2 е свързано с комплексното число 5.6–i8.3 чрез инициализация. Свързването на z1 и z3 със съответни комплексни числа може да стане чрез инициализация, подобно на z2, чрез присвояване, например z1 = z2;, или чрез задаване на стойности на полетата на променливата, например

```
z3.re = 3.4;
z3.im = -30.5;
```

свързва z3 с комплексното число 3.4 –i30.5.

Освен това, дефинирането на променлива от тип структура предизвиква отделяне на определено количество памет за всяко поле на променливата. Последното се определя от типа на полето. Полетата се разполагат последователно в паметта. Обикновено всяко поле се разполага от началото на машинна дума. Полетата, които не изискват цяло число на брой машинни думи, не използват напълно отредената им памет. Този начин за подреждане на полетата на структура се нарича **изравняване до границата на машинна дума**. За реализацията Visual C++ 6.0 размерът на 1 машинна дума е 8в.

Пример: За променливите p и q, дефинирани по-горе, ще се отделят по 16, а не по 12 байта

ОП

p.x	p.y	q.x	q.y
-	-	-2	-1.6
8в	8в	8в	8в

Допълнение: Две структури, дефинирани по един и същ начин, са различни. Например, дефинициите

```
struct str1{
   int a;
   int b;
};
```

и

```
struct str2{
   int a;
   int b;
};
```

определят два различни типа. Дефинициите

```
str1 x;
str2 y = x;
```

предизвикват грешка заради смесване на типовете (x и y са от различни типове).

Операции и вградени функции

Операциите над структури зависят от реализацията на езика. По стандарт за всяка реализация са определени следните операции и вградени функции:

а) над полетата на променливи от тип структура

Всяко поле на променлива от тип структура е от някакъв тип. Всички операции и вградени функции, допустими за данните от този тип, са допустими и за съответното поле.

б) над променливи от тип структура

- Възможно е на променлива от тип структура да се присвои стойността на вече инициализирана променлива от същия тип структура или стойността на израз от същия тип.

Пример: Допустими са присвояванията

```
z3 = z2;
p = q;
```

- Възможно е формален параметър на функция, а също резултатът от изпълнението й, да са структури. Структури с големи размери обикновено се предават чрез указатели или псевдоними на структури. Така се спестяват ресурси. Освен това, тези начини за предаване са по-сигурни.

Ще илюстрираме използването на структурите чрез следния пример.

Задача 100. да се напише програма, която:

а) въвежда факултетните номера, имената и оценките по 5 предмета на студентите от една група;

б) извежда в табличен вид въведените данни;

в) сортира в низходящ ред по среден успех данните;

г) извежда сортираните данни, като за всеки студент извежда и средния му успех.

Програма Zad100.cpp решава задачата. Данните за студент се дефинират чрез структурата:

```
struct student
{int facnom;
 char name[26];
 double marks[NUM];
};
```

където NUM е константа, определяща броя на предметите. Данните за групата са представени чрез масива

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

Реализирани са следните процедури и функции:

void read_student(student&);

Въвежда стойности на полетата на структура от тип student.

void print_student(const student &);

Извежда върху екрана полетата на структура от тип student.

void sorttable(int n, student[]);

Сортира компонентите на масив от структури от тип student в низходящ ред по среден успех. Резултатът от сортирането е в същия масив от структури.

double average(double);*

Намира средно-аритметичното на елементите на масив от NUM реални числа.

```
// Program Zad100.cpp
#include <iostream.h>
#include <iomanip.h>
#include <string.h>
const NUM = 5;
struct student
{int facnom;
 char name[26];
 double marks[NUM];
};
void read_student(student&);
void print_student(const student&);
void sorttable(int n, student[]);
double average(double*);
int main()
{cout << setprecision(2) << setiosflags(ios::fixed);
 student table[30];
 int n;
 do
 {cout << "number of students? ";
  cin >> n;
 } while (n < 1 || n > 30);
 int i;
 for (i = 0; i <= n-1; i++)
   read_student(table[i]);
 cout << "Table: \n";
 for (i = 0; i <= n-1; i++)
 {print_student(table[i]);
  cout << endl;
 }
 sorttable(n, table);
 cout << "\n New Table: \n";
 for (i = 0; i <= n-1; i++)
```

```
 {print_student(table[i]);
  cout << setw(7) << average(table[i].marks) << endl;
 }
 return 0;
}
void read_student(student& s)
{cout << "fak. nomer: ";
 cin >> s.facnom;
 char p[100];
 cin.getline(p, 100);
 cout << "name: ";
 cin.getline(s.name, 40);
 for (int i = 0; i <= NUM-1; i++)
 {cout << i << " -th mark: ";
  cin >> s.marks[i];
 }
}
void print_student(const student& stud)
{cout << setw(6) << stud.facnom << setw(30) << stud.name;
 for (int i = 0; i <= NUM-1; i++)
 cout << setw(6) << stud.marks[i];
}
void sorttable(int n, student a[])
{for (int i = 0; i <= n-2; i++)
 {int k = i;
  double max = average(a[i].marks);
  for (int j = i+1; j <= n-1; j++)
    if (average(a[j].marks) > max)
    {max = average(a[j].marks);
     k = j;
    }
  student x = a[i]; a[i] = a[k]; a[k] = x;
 }
}
double average(double* a)
{double s = 0;
```

```
    for (int j = 0; j <= NUM-1; j++)
       s += a[j];
    return s/NUM;
   }
```

Процедурата за сортиране void sorttable(int n, student a[]) е реализирана неефективно тъй като се разместват структури. При структури с големи размери сортирането е много бавно. Реализацията може да се подобри като се създаде масив от **указатели към структурите** – елементи на table. При необходимост от размяна, тя се осъществява не със структурите, а с адресите на съответните им указатели.

11.3 Указатели към структури

Дефинират се по общоприетия начин (фиг. 11.4).

Указател към структура

<указател_към_структура> ::=

struct <име_на_структура> * <променлива_указател>

[= & <променлива>]$_{опц}$;

където

<променлива> е от тип <име_на_структура>.

Фиг. 11.4 Указател към структура

В C++ запазената дума struct може да се пропусне.

Пример:

```
student st1, st2;
...
student *pst = &st1;
...
pst = &st2;
...
```

В резултат за променливата-указател pst се отделят 4В ОП, в които отначало се записва адресът на st1, след което – адресът на st2.

Достъпът до полетата на променлива от тип структура чрез указател към нея се осъществява чрез обръщението:

(*<променлива_указател>).<име_на_поле>

което е еквивалентно на

<променлива_указател> -> <име_на_поле>

За разделител са използвани знаците – и >, записани последователно.

Пример: Достъпът до полетата на st2 чрез указателя pst се реализира чрез обръщенията:

```
pst -> facnom
pst -> name
pst -> marks
```

Задача 101. Да се модифицира функцията за сортиране sorttable от задача 100, като за сортирането се използва помощен масив от указатели към структурата student.

Пред вид промени и в main ще дадем програмен фрагмент, решаващ задачата. Функциите read_student(), print_student() и average() са пропуснати, тъй като са същите като в задача 100.

```
// Program Zad101.cpp
#include <iostream.h>
#include <iomanip.h>
#include <string.h>
const NUM = 5;
struct student
{int facnom;
 char name[26];
 double marks[NUM];
};
void read_student(student&);
void print_student(const student&);
void sorttable(int n, student* []);
double average(double*);
int main()
{cout << setprecision(2) << setiosflags(ios::fixed);
 student table[30];
```

```
 student* ptable[30];
 int n;
 do
 {cout << "number of students? ";
  cin >> n;
 } while (n < 1 || n > 30);
 int i;
 for (i = 0; i <= n-1; i++)
 {read_student(table[i]);
  ptable[i] = &table[i];
 }
 cout << "Table: \n";
 for (i = 0; i <= n-1; i++)
 {print_student(table[i]);
  cout << endl;
 }
 sorttable(n, ptable);
 cout << "\n New Table: \n";
 for (i = 0; i <= n-1; i++)
 {print_student(*ptable[i]);
  cout << setw(7) << average(ptable[i]->marks) << endl;
 }
 return 0;
}
...
void sorttable(int n, student* a[])
{for (int i = 0; i <= n-2; i++)
 {int k = i;
  double max = average(a[i]->marks);
  for (int j = i+1; j <= n-1; j++)
    if (average(a[j]->marks) > max)
    {max = average(a[j]->marks);
     k = j;
    }
  student* x;
  x = a[i]; a[i] = a[k]; a[k] = x;
```

```
 }
}
...
```

За работа със структури от данни се използва подходът абстракция със структури от данни.

11.4 Абстракция със структури от данни

При този подход методите за използване на данните са разделени от методите за тяхното 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, селектори** и **предикати**, които реализират "абстрактните данни" по конкретен начин.

Ще го илюстрираме чрез следната задача.

Задача 102. Да се напише програма, която реализира основните рационално-числови операции – събиране, изваждане, умножение и деление на рационални числа.

Програма Zad102.cpp решава задачата. Тя дефинира функции за събиране, изваждане, умножение и деление на рационални числа като реализира следните общоизвестни правила:

$$\frac{n1}{d1} + \frac{n2}{d2} = \frac{n1.d2 + n2.d1}{d1.d2}$$

$$\frac{n1}{d1} - \frac{n2}{d2} = \frac{n1.d2 - n2.d1}{d1.d2}$$

$$\frac{n1}{d1} * \frac{n2}{d2} = \frac{n1.n2}{d1.d2}$$

$$\frac{n1}{d1} / \frac{n2}{d2} = \frac{n1.d2}{d1.n2}.$$

Тези операции лесно могат да се реализират ако има начин за конструиране на рационално число по зададени две цели числа, представящи съответно неговите числител и знаменател и ако има начини, които по дадено рационално число извличат неговите числител и знаменател. Затова в програмата Zad102.cpp са дефинирани функциите:

```
void makerat(rat& r, int a, int b)- която конструира рационално
                                    число r по дадени числител a и
                                    знаменател b;
int numer(rat& r)       - която намира числителя на рационалното
                          число r;
int denom(rat& r)       - която намира знаменателя на рационалното
                           число r,
```

където с rat означаваме типа рационално число.
Все още не знаем как точно да реализираме тези функции, нито как се представя рационално число, но ако ги имаме, функциите за рационално-числова аритметика и процедурата за извеждане на рационално число могат да се реализират по следния начин:

```
rat sumrat(rat& r1, rat& r2)        //събира рационални числа
{rat r;
 makerat(r, numer(r1)*denom(r2)+numer(r2)*denom(r1),
            denom(r1)*denom(r2));
 return r;
}
rat subrat(rat& r1, rat& r2)        // изважда рационални числа
{rat r;
 makerat(r, numer(r1)*denom(r2)-numer(r2)*denom(r1),
            denom(r1)*denom(r2));
 return r;
}
rat multrat(rat& r1, rat& r2)       // умножава рационални числа
{rat r;
 makerat(r, numer(r1)*numer(r2),
            denom(r1)*denom(r2));
 return r;
}
```

```
rat quotrat(rat& r1, rat& r2)       // дели рационални числа
{rat r;
 makerat(r, numer(r1)*denom(r2),
            denom(r1)*numer(r2));
 return r;
}
void printrat(rat& r)            // извежда рационално число
{cout << numer(r) << "/" << denom(r) << '\n';
}
```

Сега да се върнем към представянето на рационалните числа, а също към реализацията на примитивните операции: конструктора makerat и селекторите numer и denom. Тъй като рационалните числа са частни на две цели числа, удобно представяне на рационално число е структура от вида:

```
struct rat
{int num, den;
};
```

Тогава примитивните функции, реализиращи конструктора makerat и двата селектора numer и denom, имат вида:

```
void makerat(rat& r, int a, int b)
{r.num = a;
 r.den = b;
}
int numer(rat& r)
{return r.num;
}
int denom(rat& r)
{return r.den;
}
```

Тези функции са включени в Zad102.cpp и са използвани за намиране на сумата, разликата, произведението и делението на рационалните числа ½ и ¾.

```
// Program Zad102.cpp
#include <iostream.h>
struct rat
{int num, den;
```

```
};
void makerat(rat& r, int a, int b)
{r.num = a;
 r.den = b;
}
int numer(rat& r)
{return r.num;
}
int denom(rat& r)
{return r.den;
}
rat sumrat(rat& r1, rat& r2)
{rat r;
 makerat(r, numer(r1)*denom(r2)+numer(r2)*denom(r1),
            denom(r1)*denom(r2));
 return r;
}
rat subrat(rat& r1, rat& r2)
{rat r;
 makerat(r, numer(r1)*denom(r2)-numer(r2)*denom(r1),
            denom(r1)*denom(r2));
 return r;
}
rat multrat(rat& r1, rat& r2)
{rat r;
 makerat(r, numer(r1)*numer(r2),
            denom(r1)*denom(r2));
 return r;
}
rat quotrat(rat& r1, rat& r2)
{rat r;
 makerat(r, numer(r1)*denom(r2),
            denom(r1)*numer(r2));
 return r;
}
void printrat(rat& r)
```

```
{cout << numer(r) << "/" << denom(r)<< endl;
}
int main()
{rat r1, r2;
 makerat(r1, 1, 2);
 makerat(r2, 3, 4);
 printrat(sumrat(r1, r2));
 printrat(subrat(r1, r2));
 printrat(multrat(r1, r2));
 printrat(quotrat(r1, r2));
 return 0;
}
```

Реализирането на подхода абстракция със структури от данни в Задача 102, показва следните четири нива на абстракция:

- Използване на рационалните числа в проблемна област (във функцията main);
- Реализиране на правилата за рационално-числова аритметика (sumrat, subrat, multrat, quotrat, printrat);
- Избор на представяне на рационалните числа и реализиране на примитивни конструктори и селектори (makerat, numer и denom);
- Работа на ниво структура.

Използването на подхода прави програмите по-лесни за описание и модификация. Ако разгледаме по-внимателно изпълнението на горната програма, забелязваме, че тя има редица недостатъци, но основният е, че не съкращава рационални числа. За да поправим този недостатък, се налага да променим единствено функцията makerat. За целта ще използваме помощната функция gcd, дефинирана в глава 8. Новата makerat има вида:

```
void makerat(rat& r, int a, int b)
{if (a == 0) {r.num = 0; r.den = b;}
 else
 {int g = gcd(abs(a), abs(b));
  if (a > 0 && b > 0 || a < 0 && b < 0)
  {r.num = abs(a)/g; r.den = abs(b)/g;}
  else {r.num = - abs(a)/g; r.den = abs(b)/g;}
```

```
 }
}
```

С това проблемът със съкращаването на рационални числа е решен. За разрешаването му се наложи малка модификация на програмата, която засегна само премитивния конструктор makerat. Последното илюстрира лесната модифицируемост на програмите, реализиращи горния подход.

11. 5 От структури към класове

В тази задача дефинирахме структурата rat, определяща рационални числа, и реализирахме някой основни функции за работа с такива числа. *Възниква въпросът:* Може ли да използваме тази структура като тип данни рационално число? Отговорът е не, защото при типовете данни представянето на данните е скрито за потребителя. На последния са известни множеството от стойности и операциите и вградените функции, допустими за типа. Така възниква усещането, че трябва да се обедини представянето на рационално число като запис с две полета с примитивните операции (makerat, numer и denom), пряко използващи представянето. Последното е възможно, тъй като в езика C++ се допуска полетата на структура да са функции, разбира се от тип, различен от типа на структурата.

В следващото описание примитивните операции, реализирани чрез функциите makerat, numer и denom, а също функцията за извеждане на рационално число, ще направим полета на структутара rat. За целта реализираме следните две стъпки:

• Включване във фигурните скоби на дефиницията на структурата rat на декларациите:

```
void makerat(rat& r, int a, int b);
int numer(rat& r);
int denom(rat& r);
void printrat(rat& r);
```

от които елеминираме участието на формалния параметър rat& r, тъй като неговата функция ще се изпълни от полетата num и den. Получаваме структурата:

```
struct rat
{int num;
```

```
 int den;
 void makerat(int a, int b);
 int numer();
 int denom();
 void printrat();
};
```

която може да се интерпретира като запис с две полета num и den, над които могат да се изпълняват функциите:

- makerat, която конструира рационалното число a/b чрез свързването на num и den с a и b съответно;
- numer, която намира числителя num на рационалното число num/den;
- denom, която намира знаменателя den на рационалното число num/den;
- printrat, която извежда рационалното число num/den.

• Отразяване на тези промени в дефинициите на функциите. За целта ще изтрием участията на rat& r и r., а между типа и името на всяка функция ще поставим името на структурата rat, следвано от оператора ::.

Получаваме:

```
void rat::makerat(int a, int b)
{if (a == 0) {num = 0; den = b;}
 else
 {int g = gcd(abs(a), abs(b));
  if (a > 0 && b > 0 || a < 0 && b < 0)
  {num = abs(a)/g;
   den = abs(b)/g;
  }
  else
   {num = - abs(a)/g;
    den = abs(b)/g;
   }
 }
}
int rat::numer()
```

```
{return num;
}
int rat::denom()
{return den;
}
void rat::printrat()
{cout << num << "/" << den << endl;
}
```

Тези функции се наричат **член-функции на структурата rat**. Извикването им се осъществява като полета на структура.

Пример:

```
rat r;                // дефиниция на променлива от тип rat
r.makerat(1, 5)       // r се свързва с рационалното число 1/5
r.numer()             // намира числителя на r, в случая 1
r.denom()             // връща знаменателя на r, в случая 5
r.printrat()          // извежда върху екрана r.
```

Обръщението r.numer() е еквивалентно на изпълнение на оператора
`return r.num;`

Програма Zad102_1.cpp реализира последните промени.

```
Program Zad102_1.cpp
#include <iostream.h>
#include <math.h>
struct rat
{int num;
 int den;
 void makerat(int, int);
 int numer();
 int denom();
 void printrat();
};
void rat::printrat()
{cout << num << "/" << den << endl;
}
```

```
int gcd(int a, int b)
{while (a!=b)
 if (a > b) a = a-b; else b = b-a;
 return a;
}
void rat::makerat(int a, int b)
{if (a == 0) {num = 0; den = b;}
 else
 {int g = gcd(abs(a), abs(b));
  if (a>0 && b>0 || a<0 && b < 0)
  {num = abs(a)/g;
   den = abs(b)/g;
  }
  else
  {num = - abs(a)/g;
   den = abs(b)/g;
  }
 }
}
int rat::numer()
{return num;
}
int rat::denom()
{return den;
}
rat sumrat(rat& r1, rat& r2)
{rat r; r.makerat(r1.numer() * r2.denom() +
                  r2.numer() * r1.denom(),
                  r1.denom() * r2.denom());
 return r;
}
rat subrat(rat& r1, rat& r2)
{rat r; r.makerat(r1.numer() * r2.denom() -
                  r2.numer() * r1.denom(),
                  r1.denom() * r2.denom());
 return r;
```

```
}
rat multrat(rat& r1, rat& r2)
{rat r; r.makerat(r1.numer()*r2.numer(),
                  r1.denom()*r2.denom());
 return r;
}
rat quotrat(rat& r1, rat& r2)
{rat r; r.makerat(r1.numer()*r2.denom(),
                  r1.denom()*r2.numer());
 return r;
}
int main()
{rat r1; r1.makerat(-1,2);
 rat r2; r2.makerat(3,4);
 sumrat(r1, r2).printrat();
 // или rat r = sumrat(r1, r2); r.print();
 subrat(r1, r2).printrat();
 // или r = subrat(r1, r2); r.printrat();
 multrat(r1, r2).printrat();
 // или r = multrat(r1, r2); r.printrat();
 quotrat(r1, r2).printrat();
 // или r = quotrat(r1, r2); r.printrat();
 return 0;
}
```

Забелязваме, че във функциите sumrat, subrat, multrat и quotrat не се използват полетата на записа num и den, но ако направим опит за използването им даже на ниво main, опитът ще бъде успешен. Последното може да се забрани, ако се използва етикетите private: пред дефиницията на полетата num и den и public: пред декларациите на член-функциите. Структурата rat получава вида:

```
struct rat
{private:
 int num;
 int den;
 public:
 void makerat(int, int);
```

```
 int numer();
 int denom();
 void printrat();
};
```

Опитът за използването на полетата num и den на структурата rat извън член-функциите вече ще предизвиква грешка.

Етикетите private и public се наричат **спецификатори за достъп**. Всички член-данни, следващи спецификатора за достъп private, са достъпни само за член-функциите от дефиницията на структурата. Всички член-данни и член-функции, следващи спецификатора за достъп public, са достъпни за всяка функция, която е в областта на структурата. Ако специфитаторите за достъп с пропуснати, всички членове са public. Един и същ спецификатор за достъп може да се използва повече от веднъж в една и съща дефиниция на структура.

Така специфицирането на num и den като private прави невъзможно използването им извън член-функциите makerat, numer, denom и printrat.

Ако заменим запазената дума struct със class, последната програма не променя поведението си. Така дефинирахме първия си клас с име rat, създадохме два негови обекта – рационалните числа r1 и r2 и работихме с тях.

Задача 103. Като се използва подходът абстракция със структури от данни да се даде ново представяне на задача 100.

```
// Program Zad103
#include <iostream.h>
#include <iomanip.h>
#include <string.h>
const NUM = 5;
class student
{private:
 int facnom;
 char name[26];
 public:
 double marks[5];
```

```
 void read_student();
 void print_student();
};
void sorttable(int n, student[]);
double average(double*);
int main()
{cout << setprecision(2) << setiosflags(ios::fixed);
 student table[30];
 int n;
 do
 {cout << "number of students? ";
  cin >> n;
 }while (n < 1 || n > 30);
 int i;
 for (i = 0; i <= n-1; i++)
   table[i].read_student();
 cout << "table: \n";
 for (i = 0; i <= n-1; i++)
 {table[i].print_student();
  cout << endl;
 }
 sorttable(n, table);
 cout << "\n New Table: \n";
 for (i = 0; i <= n-1; i++)
 {table[i].print_student();
  cout << setw(7) << average(table[i].marks) << endl;
}
 return 0;
}
void student::read_student()
{cout << "fak. nomer: ";
 cin >> facnom;
 char p[100];
 cin.getline(p, 100);
 cout << "name: ";
 cin.getline(name, 40);
```

```
 for (int i = 0; i <= NUM-1; i++)
 {cout << i << " -th mark: ";
  cin >> marks[i];
 }
}
void student::print_student()
{cout << setw(6) << facnom << setw(30) << name;
 for (int i = 0; i <= NUM-1; i++)
 cout << setw(6) << marks[i];
}
void sorttable(int n, student a[])
{for (int i = 0; i <= n-2; i++)
 {int k = i;
  double max = average(a[i].marks);
  for (int j = i+1; j <= n-1; j++)
    if (average(a[j].marks) > max)
    {max = average(a[j].marks);
     k = j;
    }
  student x = a[i]; a[i] = a[k]; a[k] = x;
 }
}
double average(double* a)
{double s = 0;
 for (int j = 0; j <= NUM-1; j++)
    s += a[j];
 return s/NUM;
}
```

Тези идеи са в основата на нов подход за програмиране – обектно – ориентирания.

Задачи

Задача 1. Да се напише функция, която намира разстоянието между две точки в равнината. Като се използва тази функция, да се напише програма, която въвежда координатите на n точки от равнината, намира и извежда най-голямото разстояние между тях. За целта да се дефинира структура, определяща точка от равнината с координати (x, y).

Задача 2. Да се напише програма, която въвежда факултетните номера, имената и успеха по k предмета на студентите от една група и извежда следната таблица:

N	име	предмет1	...	предметK	среден успех
.	.	.	.	.	
.	.	.	.	.	
.	.	.	.	.	
		ср. успех	...	ср. успех	ср. успех

Задача 3. Да се напише:

а) булева функция equal(rat x, rat y), която установява дали рационалните числа x и y са равни.

б) булева функция grthen(rat x, rat y), която установява дали рационалното число x е по-голямо от рационалното число y.

в) функция maxrat(int n, rat x[]), която намира най-голямото от рационалните числа на масива x.

г) функция sortrat(int n, rat x[]), която сортира елементите на редицата от рационални числа x_0, x_1, ..., x_{n-1}.

Задача 4. Да се напише програма, която решава системата уравнения

$$a\,x + b\,y = e$$
$$c\,x + d\,y = f$$

където коефициентите a, b, c, d, e, f, а също и неизвестните x и y са рационални числа.

Задача 5. Нека a_0, a_1, ..., a_{n-1} и x са рационални числа. Да се напише функция, която намира стойността на полинома

$P(x) = a_0x^n + a_1x^{n-1} + ... + a_n$.

Задача 6. Да се дефинира структура, определяща точка от равнината с координати (x, y), където x и y приемат за стойности числата от 1 до 100. Да се напише програма, която чете координатите на четири точки, представляващи върховете A, B, C и D на четириъгълник в

цикличен ред и определя дали ABCD е квадрат, правоъгълник или друга фигура.

Задача 7. Да се напише програма, която решава системата уравнения

$$a\,x + b\,y = e$$
$$c\,x + d\,y = f$$

където коефициентите a, b, c, d, e, f, а също и неизвестните x и y са комплексни числа.

Задача 8. Нека a_0, a_1, ..., a_{n-1} и x са комплексни числа. Да се напише функция, която намира стойността на полинома

$P(x) = a_0x^n + a_1x^{n-1} + ... + a_n$.

Задача 9. Дадени са естественото число n и комплексното число z. Да се напише програма, която пресмята стойността на следната комплексна функция:

$$F1 = 1 + \frac{z}{1!} + \frac{z^2}{2!} + ... + \frac{z^n}{n!}$$

Допълнителна литература

1. B. Stroustrup, C++ Programming Language. Third Edition, Addison – Wesley, 1997.
2. Ал Стивънс, Кл. Уолнъм, C++ библия, АЛЕКС СОФТ, София, 2000.
3. М. Тодорова, Езици за функционално и логическо програмиране – функционално програмиране, СОФТЕХ, София, 1998.
4. Ст. Липман, Езикът C++ в примери, КОЛХИДА ТРЕЙД КООП, София, 1993.
5. Ч. Сфар, Visual C++ 6.0, том 1, СОФТПРЕС, София 2000.