Записки по Обектно-Ориентирано Програмиране (ООП) за специалности Приложна Математика, Статистика, Информационни Системи и Информатика

Преподавател: доц. д-р Павел Павлов

1. Структури. Основни операции

Като първа стъпка при структуриране на входната информация за дадена програма най-често ние се опитваме да групираме логически свързаните данни в едно цяло. Обикновено данните, влизащи в една такава група, се разполагат последователно в паметта на компютъра. Пример за такова подреждане са масивите, но има едно съществено ограничение: всички елементи на масива задължително трябва да са данни от един и същ тип. В много ситуации се налага да групираме данни от различен тип. За тази цел в C е въведен специален тип данни, наречен структура. Структурата е наредена съвкупност от краен брой елементи от различен тип, която има йерархична (дървовидна) организация. Структурата позволява обединяването на прости данни от различен тип и записването им под едно име.

Описанието на данни, които са структура, започва със служебната дума struct и има следния общ вид:

```
struct име_на_структурата
{
  описание_на_първия_елемент_от_данни;
  ...
  описание_на_последния_елемент_от_данни;
} ;
```

където име_на_структурата (тип на структурата) е произволен идентификатор, валиден за езика.

Данните, описани в тялото на структурата, се наричат елементи (полета от данни). Те могат да бъдат от произволен, допустим за езика тип и се описват по установения начин.

Ще разгледаме като прост пример на структура следното приложение: в програмата е необходимо многократно да използваме текущото време, записано като часове, минути и секунди. Секундите се измерват с дробна част.

Вместо да се въведат три отделни променливи за часовете, минутите и секундите, много по-правилно е да се използва структура от вида:

```
struct time
 {  int hour; // променлива за часовете
    int min;  // променлива за минутите
    float sec;  // променлива за секундите
 } ;
```

Описанието на структура не определя конкретна променлива, която да представя дадената структура. Описанието определя само конкретен тип структура, т.е. един шаблон или модел, който по-късно може да бъде приписан на една или повече променливи. Ето защо при компилиране на описанието на типа структура не се резервира памет за елементите (полетата) и това е едно от предимствата при работа със структури. Една или повече променливите могат да се дефинират като структура от определен тип по два начина:

- променливите се изброяват като списък (разделени със запетаи) след затварящата скоба в описанието на структурата:

```
struct time
{ int hour;
  int min;
  float sec;
} u, v, t;
```

Тук u, v, t са променливи от тип структура (структурни променливи).

- в произволно място на програмата (но допустимо за описания) се дефинират новите променливи u, v, t чрез описание от вида:

```
struct time
{ int hour;
  int min;
  float sec;
} ;
struct time u, v, t;
```

Тук дефинираме променливи u, v, t от тип структура time.

Ако всички променливи са дефинирани в описанието на структурата, името на структурата може да се изпусне:

```
struct
{ int hour;
  int min;
  float sec;
} u, v, t;
```

Ако се наложи обаче допълнително да се дефинира променлива от същия тип структура, трябва да се повтори цялото описание на структурата.

В C++ резервираната дума struct може да се изпусне в дефиницията на структурни променливи, например time u, v, t;, но в C не може.

Името на типа на структурата и имената на полетата на структурата от една страна и имената на произволни променливи от друга страна могат да съвпадат. Имената на отделните полета трябва да са различни помежду си.

Полетата на дадена структура се разполагат последователно в оперативната памет. При обработване на дефиниция на структурна променлива, компилаторът ще разпредели памет, достатъчна за да могат да се запишат всичките полета на структурата.
Броят на байтовете, които се разпределят за една променлива от тип структура не е винаги равен на сумата от дължините на отделните полета, заради влиянието на друг фактор, който се нарича **подравняване на данните** – Structure Alignment. Подравняването се задава от опциите на компилатора. При 32-битовите компилатори има възможност за подравняване по дума (32 бита) или по двойна дума (64 бита). Ако подравняването е по граници на байт, тогава полетата са разположени едно след друго и дължината на структурата ще е сума от дължините на отделните полета.

При друго подравняване компилаторът разполага между отделните полета на една структурна променлива и между елементите на масив празни байтове, така че да се изпълняват следните условия:

- отделната структурна променлива, в частност когато тя е елемент на масив, да започва на границата на дума, двойна дума или четворна дума. Това значи, че тази граница трябва да се дели на 2, на 4 или на 8;
- всеки тип, различен от char, да започва на границата на дума – двойна или четворна;
- при необходимост в края на структурата се добавят празни байтове, така че общият брой байтове на структурата да е кратен на 2, 4 или на 8.

Точният брой байтове на една структура може да се получи с операцията sizeof, например:
sizeof (struct time) – при 16-битов компилатор, hour, min заемат по 2 байта, sec заема 4 байта. Ако подравняването е по дума, структурата ще заема 8 байта.

Ако подравняването е по двойна дума, в края на hour и в края на min ще се добавят по два празни байта.
Цел на Structure Alignment – да се даде възможност за оптимизация; ако за потребителя е важна икономията на памет, тогава може да се използва подравняване по байтове. Ако потребителят търси по-голяма ефективност, тогава може да се използва подравняването по двойна или четворна дума – остават свободни байтове и се хаби памет, но реализацията е по-бърза.

Основни операции със структури

Прилагане на адресна операция & – определя се адреса на една структурна променлива и няма разлика от адресна операция с обикновените променливи. Единствена особеност има в описанието на указателите, на които евентуално бихме искали да присвоим този адрес. Нека да използваме описанието на структурата time и да разгледаме следния фрагмент от програма:

.....
struct time *p; // p е указател към структура от тип time
p = &u; // На p се присвоява адреса на u
След като е определен адресът на една структура, той може да бъде предаден като фактически параметър на функция, така както се предават указатели.
Основна операция с дадена структура е **обръщение към (използване на) полетата на структурата**. Тя се осъществява по два начина:

- чрез съставен идентификатор като се използва операцията точка '**.**'

Синтаксис:
идентификатор_на_променливата_структура**.**идентификатор_на_поле

Пример:

```
struct time x;
int k, n;
```

.....
n = x.hour; // x.hour – полето hour от структурната променлива x
k = x.min;
.....

Тази операция можем да я използваме с псевдоними:

struct time &newx = x;
n = newx.hour;
k = newx.min;

Стойностите на k и n ще са същите както по-горе.

- чрез описан указател p към структурата, като използваме указателната операция, която се записва като знак минус, следван от знака по-голямо

Синтаксис:

p-> идентификатор_на_поле

Пример:

.....
struct time *p;
int k, n;

```
p = &u;
.....
n = p ->hour;
k = p ->min;
.....
```

Пример: ако едномерния масив q е елемент на структурата t, обръщението към q се осъществява по следния начин:

```
.....
int k;
……..
k = t.q[5];
.....
```

На променливата k е присвоен шестият елемент на масива q.
Операциите ‘.’ и ‘->’ са взаимозаменяеми.
Например вместо

```
n = x.hour;
```

можем да запишем

```
n = (&x)->hour;
```

а к = p->hour;
е еквивалентно на
к = (*p).hour;

Операцията ‘**.**’ осъществява пряк достъп до поле на една структурна променлива, а операцията ‘->’ осъществява косвен достъп. И двете операции имат приоритет 1 (най-високия) и са ляво-асоциативни. По тази причина слагаме скоби.

Инициализиране на структури. Променливите от тип структура могат да бъдат инициализирани както другите типове променливи. След дефинирането на променливата се поставя знак за присвояване и в големи скоби се изброяват началните стойности на полетата от структурата. Началните стойности се присвояват в реда в който са описани.
Пример:
static struct time example = { 5, 55, 25.66};

Едновременно с дефинирането на променливата example като структура от тип time на нейните елементи се присвояват начални стойности.

Могат да се инициализират само променливи – структури, които са дефинирани в клас памет static или extern. Както при масивите началните стойности на променлива static трябва да са константи или изрази от константи. Ако не сме задали стойности за всички полета, останалите получават стойност 0. Ако началните стойности са повече от полетата се извежда съобщение за грешка. Със структурни променливи от един и същи тип може да се извърши операция присвояване една структурна променлива на друга.

Например:
u = v; //при горните дефиниции
При такава операция стойностите на полетата на структурната променлива отдясно се присвояват на полетата на променливата отляво. По-точно се осъществява физическо копиране на толкова байтове, колкото е размера на съответната структура.

2. Вложени структури. Рекурсивно използване на структури

В дефиницията на структура няма ограничения за типа на нейните елементи. Следователно даден елемент на структура може да бъде също структура. Такива структури са вложени една в друга. Обръщението към елемент на вложена структура се осъществява чрез последователното прилагане на операциите . или ->. Редът на изпълнение на операциите е отляво надясно.

Да разгледаме следното описание:

```
struct asoc
{int k;
char s;
struct org pri;
float x;
} exa;
```

Променливата pri е описана като структура от тип org и е вложена в структура от тип asoc. Предполага се, че типът struct org е описан преди struct asoc. Записът

exa.pri.fs

е обръщение към елемента fs на структурата pri, вложена в структурата exa, в което се използва операцията точка. Ако p е указател към структурата pri, а q указател към структурата exa, обръщението може да се реализира с операцията -> по следния начин:

q->p->fs

Има възможност самата дефиниция да е вложена:

```
struct book
{ char name[40];
  struct person
   { char name[30];
     int born, died;
   } author;
  int year;
  float price;
}
```

Рекурсивно използване на структури

Поле на структура може да бъде указател. Това е напълно допустима езикова конструкция, която е в съответствие с определението за структура.

Да разгледаме следното описание:

```
struct point
 { int k;
   int *p;
   char *q[5];
 } ;
```

Да дефинираме променливата pr като структура от тип point

```
struct point pr; // дефиниране на структурна променлива от тип point
```

Обръщението към полетата на структурата pr, които са указатели, се осъществява по общоприетия начин: pr.p, pr.q[1];. Те могат да участват във всички допустими адресни операции. Например променливата, която сочи p, е *pr.p. Не е нужно да се записва *(pr.p), защото операция ‘.’ е с по-голям приоритет от адресната операция *.

Типът на указателите, описани в тялото на структурата, може да е всеки допустим тип, включително и структура. Логично следва въпросът: възможно ли е елемент на дадена структура, който е указател, да сочи към структура от същия тип? Отговорът е положителен, но се нуждае от уточняване. C и C++ не допускат поле от една структура да бъде структура от същия тип. С други думи, преди да е завършено описанието на дадена структура, на нея не могат да й се приписват променливи. Тъй като указателят не дефинира променлива, а само сочи към такава, за него това ограничение не е в сила. Затова в C са допустими конструкции от вида:

```
struct linkedlist
 { int field;
   struct linkedlist *next;
 } ;
```

Указателят next, описан в тялото на структурата linkedlist, сочи към променлива – структура от тип linkedlist. В такъв случай говорим за **рекурсивно използване** на структура.

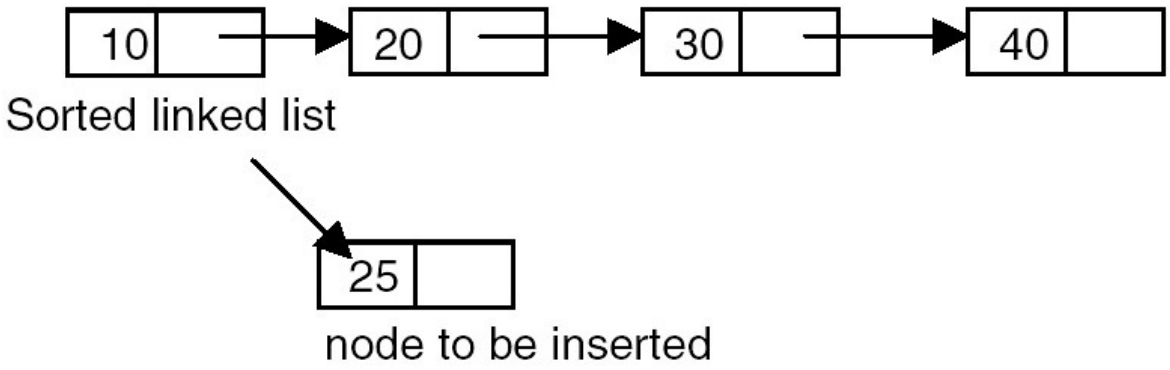
Чрез рекурсивно използване на структури могат да се съставят сложни описания на данни като линейни списъци и дървовидни структури. Линейният списък е структура от данни, в която всеки елемент от данните сочи към следващия. Предимствата на линейния списък проличават добре при решаването на задачата за сортировка, която се състои в следното:

- набор от данни трябва да се подреди по отношение на даден признак;
- при постъпването на нови данни в набора те трябва да се запишат на съответното място в подредбата съгласно с признака.

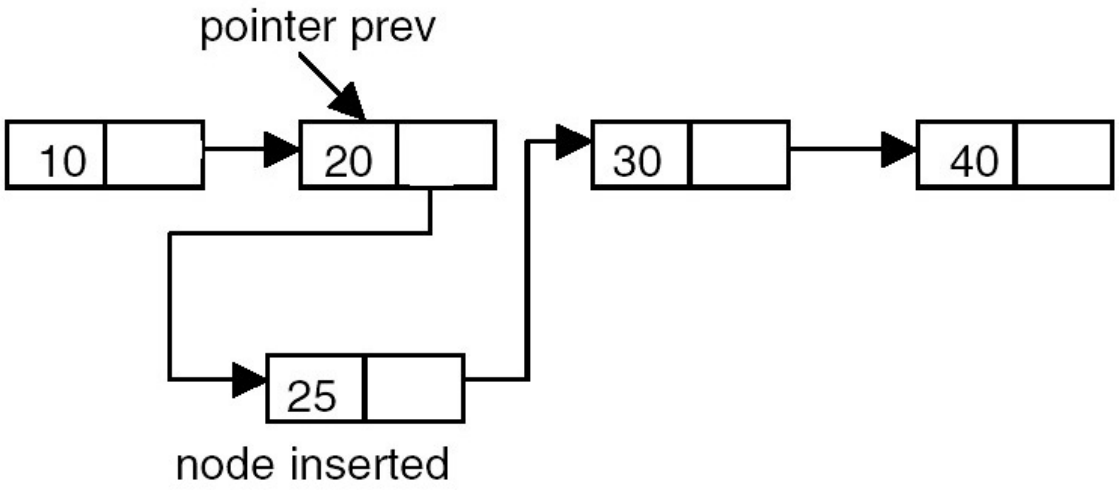
Ако използваме масив, решението има недостатъци. С помощта на рекурсивни структури набор от данни може да бъде ефективно реализиран като линеен списък.

Пример: Да се напише програма за работа с подреден в нарастващ ред линеен свързан списък от цели положителни числа. Да се напишат функции за:

1. добавяне на нов елемент в списъка, така че да се запази наредбата;
2. изтриване на елемент с дадена стойност.



Before insertion



After insertion

```
#include <iostream.h>
#include <stdlib.h>
#include <stdio.h>
struct listNode {int data;
          struct listNode *nextPtr;};
listNode *startPtr = NULL;
/* Print the instructions */
void instructions()
{ cout << "Enter your choice:\n"
          "   1 to insert an element into the
list.\n"
          "   2 to delete an element from the
list.\n"
          "   3 to end.\n";
}
```

```
/* Insert a new value into the list in sorted order
*/
void insert( int value)
{listNode *newPtr, *previousPtr, *currentPtr;
 newPtr = new listNode;
 if (newPtr != NULL) {    /* is space available */
   newPtr->data = value;
   newPtr->nextPtr = NULL;
    previousPtr = NULL;
   currentPtr = startPtr;
    while (currentPtr != NULL && value > currentPtr-
>data) {
        previousPtr = currentPtr;          /* walk to
...   */
        currentPtr = currentPtr->nextPtr;  /* ...
next node */
     }
   if (previousPtr == NULL) {
```

```
      newPtr->nextPtr = startPtr;
         startPtr = newPtr;
      }
      else {
         previousPtr->nextPtr = newPtr;
         newPtr->nextPtr = currentPtr;
      }
   }
   else
     cout << value << " not inserted. No memory
available.” << endl;
}
```

```
/* Delete a list element */
int delete1(int value)
{listNode *previousPtr, *currentPtr, *tempPtr;
  if (value == startPtr->data) {
   tempPtr = startPtr;
   startPtr = startPtr->nextPtr;  /* de-thread the
node */
   free(tempPtr);              /* free the de-threaded
node */
   return value;
  }
 else {
   previousPtr = startPtr;
   currentPtr = startPtr->nextPtr;
   while (currentPtr != NULL && currentPtr->data !=
value) {
        previousPtr = currentPtr;          /* walk to
...   */
```

```
         currentPtr = currentPtr->nextPtr;  /* ...
next node */
      }
   if (currentPtr != NULL) {
         tempPtr = currentPtr;
         previousPtr->nextPtr = currentPtr->nextPtr;
         free(tempPtr);
       return value;
      }
  }
 return 0;
}
```

```
/* Return 1 if the list is empty, 0 otherwise */
int isEmpty()
{
   return startPtr == NULL;
}
/* Print the list */
void printList(listNode *currentPtr)
{if (currentPtr == NULL) cout << "List is empty.” <<
endl << endl;
   else {
    cout << "The list is:” << endl;
    while (currentPtr != NULL) {
       cout << currentPtr->data << “ --> ";
       currentPtr = currentPtr->nextPtr;
     }
    cout << "NULL” << endl << endl;
 }
}
```

```
void main()
{int choice; int item;
 instructions();  /* display the menu */
 cout << "? "; cin >> choice;
 while (choice != 3) {
   switch (choice) {
      case 1:
         cout << "Enter a number: "; cin >> item;
         insert(item);
         printList(startPtr);
         break;
      case 2:
         if (!isEmpty()) {
            cout << "Enter number to be deleted: " <<
endl;
            cin >> item;
            if (delete1(item)) {
               cout << item << " deleted" << endl;
```

```
                printList(startPtr);
             }
             else
                cout << item << " not found" << endl <<
endl;
          }
          else
             cout << "List is empty." << endl << endl;
          break;
       default:
          cout << "Invalid choice." << endl << endl;
          instructions();
          break;
    }
    cout << "? "; cin >> choice;
  }
cout << "End of run." << endl;
}
```

3. Функции и структури

При съвместната работа с функции и структури са допустими следните действия:

1. Структура може да се дефинира в тялото на функция, като се използва модел на структура, определен извън функцията на глобално ниво.
2. Указател към структура може да се предаде като параметър на функция. По този начин се осигурява достъп в тялото на функцията до полетата на структурна променлива, дефинирана извън функцията. Обръщението към самите полета се осъществява с указателната операция ‘->’.
3. Върнатата величина от една функция може да е указател към структура.

Пример: Дефинират се две структури ksi и eps от тип type. Едновременно с дефинирането структурите се инициализират. Елементът a на структурите е масив с 5 елемента от тип int. От двете структури се определя тази, която е с по-голяма сума на елементите на масива a. На екрана се извежда стойността на втория елемент к на така определената структура.

```
#include <iostream.h>
struct type
 { int a[5];
   int k;
 } ksi = { 1, 6, 3, 7, 2, 1}, eps = { 2, 8, 9, 0, -
5, 2};
void main ()
 { struct type *p = &ksi, *q = &eps;
   struct type *maxi (struct type *, struct type *);
   // функцията maxi има параметри два указателя към
структурата
   // type и връща указател към тази структура;
   cout << " Стойността на k е: " << maxi(p, q)->k
<< endl;
 }
```

```
struct type *maxi (struct type *u, struct type *v)
 { int i, s1 = 0, s2 = 0;
   for (i = 0; i < 5; i++)
    { s1 += u->a[i];
      s2 += v->a[i];
    }
  if (s1>s2) return u;
  else return v;
 }
```

В тази програма, ако променим формалния параметър, това ще промени и съответния фактически параметър.

Изложеният начин за предаване на структури като параметри (чрез указатели) е универсален – той се поддържа в K&R C, в ANSI C и в C++.

В ANSI C и в C++ може директно да се предава структура като параметър на функция. Формалният параметър се описва в заглавието като структура. Обръщението към полетата на структурата в тялото на функцията се осъществява с операцията ‘.’. В ANSI C и C++ структура може да бъде върната като резултат от изпълнението на функция. В този случай типът на връщания резултат се описва като структура:
struct тип_на_структурата име_на_функцията();
Извикването на функция с параметър структура е свързано с допълнителен разход на време и памет, защото се записва копие на цялата структура в програмния стек. При това промяната на копието не влияе на структурата, която е фактически параметър. По същата причина, когато връщаната стойност на една функция е структура, се губи време и памет за записване в програмния стек на копие на тази структура.
Ако масив е поле на структура, то предавайки структурата по стойност ние предаваме и масива по стойност – това е начинът за предаване на масив по стойност.

Примерна програма (с предаване по стойност):

```
#include <iostream.h>
struct type
 { int a[5];
   int k;
 };
void main ()
 { struct type ksi = { 1, 6, 3, 7, 2, 1}, eps = { 2,
8, 9, 0, -5, 2} ;
   struct type maxi (struct type, struct type);
   cout << " Стойността на k е: " << maxi(ksi,
eps).k << endl;
 }
```

```
struct type maxi (struct type u, struct type v)
 { int i, s1 = 0, s2 = 0;
   for (i = 0; i < 5; i++)
    { s1 += u.a[i];
      s2 += v.a[i];
    }
  if (s1>s2) return u;
  else return v;
 }
```

В тази реализация, ако променим формалния параметър, това не засяга съответния фактически параметър.

Само в C++ има още един начин за предаване на структура като параметър и връщане на структура като резултат от функция. В C++ може да се предава структура като параметър на функция, като формалният параметър се опише като псевдоним. Обръщението към полетата на структурата в тялото на функцията се осъществява с операцията ‘**.**’. Също така само в C++ име на структура може да бъде върнато като резултат от функция. В този случай типът на връщания резултат се описва като псевдоним.

```
#include <iostream.h>
struct type
 { int a[5];
   int k;
 };
void main ()
 { struct type ksi = { 1, 6, 3, 7, 2, 1}, eps = { 2,
8, 9, 0, -5, 2} ;
   struct type &maxi (struct type &, struct type &);
   cout << " Стойността на k е: " << maxi(ksi,
eps).k << endl;
 }
```

```
struct type &maxi (struct type &u, struct type &v)
 { int i, s1 = 0, s2 = 0;
   for (i = 0; i < 5; i++)
    { s1 += u.a[i];
      s2 += v.a[i];
    }
  if (s1>s2) return u;
  else return v;
 }
```

Съпоставяне на трите подхода:

- с указатели - поддържа се от K&R C, ANSI C, C++; предимства – оптимален по време и памет; недостатъци – работи с указатели;
- с предаване по стойност (директно) – поддържа се от ANSI C и C++; предимства – по-естествен запис, по-просто за програмиране; недостатъци – разход на време и памет;
- с псевдоними – поддържа се от C++; обединява предимствата на предните два подхода.

4. Структури и масиви

Елементите на един масив могат да бъдат данни от произволен тип, в частност те могат да бъдат структури. Описанието на масив, чиито елементи са структури, е от вида:
struct тип_на_структурата идентификатор_на_масива [];
Например:

```
struct type
 { int k;
   int l;
 };
```

struct type a[5]; //a – масив с пет елемента от структурен тип type.
Еквивалентна форма на горната дефиниция:

```
struct type
 { int k;
   int l;
 } a[5];
```

Обръщението към полетата на структура, която е елемент на масив, е следното: a[1].k – полето k в структурната променлива (елемент от масив) a[1]; аналогично a[1].l е полето l на a[1].
С това обръщение се избира елементът k или l на структурата, която е втори елемент на масива a. Така означените полета се разглеждат като обикновени променливи. Типът им се определя съгласно с описанието в тялото на структурата. Те могат да участват в изрази, да се предават като параметри във функции и т.н. Например следните оператори са напълно валидни:

```
a[1].k = 5;
a[1].k = a[1].l + 2;
++a[1].k;
```

Достъпът до елементите на масива може да се осъществява и чрез използването на името на масива, което е указател-константа към първия елемент на масива.
Например:
(*(a+1)).k ⇔ a[1].k ⇔ (a+1)->k (използваме операция ‘->’)

Можем да използваме указател променлива, като го инициализираме с адреса на първия елемент от масива.
Например:
struct type *ptr;
ptr = a;
(ptr+1) сочи към a[1], (ptr+1)->k е полето k на структурната променлива a[1].
Изменението на указателя с операцията ++ (--) съответно го увеличава (намалява) така, че да сочи следващия (предишния) елемент от масива със структурните променливи.
Масив от структури може да бъде предаван като параметър на функция. В заглавието на функцията той трябва да се декларира, че е от тип ” struct тип_на_структурата”, а типът на структурата да бъде описан на глобално ниво:
void fun (int x, struct type a[]) // описанието на структурата type е глобално за
// функцията fun
{ …

}
Можем да имаме следното обръщение:
fun(5, a);
В ANSI C и C++ елемент на масив от структури може да се предава директно като параметър на функция. Например за:
void trick(struct type x)
{ …
}
можем да имаме следното обръщение:
trick(a[2]);
където a е масив от структури.
В K&R C, ANSI C и C++ ние можем да предадем указател към елемент на масив като параметър на функция. Например за:
void trick (struct type *p)
{ …
}
можем да има следното обръщение:
trick (a + 2);

където a е масив от структури.
Само в C++ елементите на масив от структури могат да се предават по име чрез псевдоним. Например за:

```
void trick (struct type &x)
 { …
 }
```

можем да имаме следното обръщение:

```
trick (a[2]);
```

където a е масив от структури.
Ако елемент от структура е масив и структурата също е елемент на масив, възможни са обръщения от вида a[1].b[2], както в следния пример:

```
struct type
 { int b[3];
   int k;
 } a[5];
```

До елементите на масив, който е елемент на структура, достигаме по обичайния начин:

```
int x;
.....
x = a[3].b[1];
```

Масив от структури се инициализира по общоприетия начин с тази особеност, че началните стойности за всеки елемент на масива може да се заграждат в големи скоби, например:

```
static struct type c[2] = { { 1, 2}, { 3, 5} };
```

В сила са същите ограничения както при инициализиране на обикновените масиви.

Пример: Даден е масивът a, чиито елементи са структури от тип array. Необходимо е да се сортират елементите на масива по отношение на елемента к на всяка структура във възходящ ред. След сортирането в новото подреждане се търси първия положителен елемент x на структурите.

```
#define LIMIT 5
struct array
 { float x;
   int k;
 } a[LIMIT];
```

Реализираме метода на бързото сортиране – идеята се състои в намиране на онова място на даден елемент x от масива, което той би заел, ако масивът е сортиран. В процеса на търсене на мястото на елемента x, останалите елементи от масива се разменят по такъв начин, че след завършване на функцията, елементите, които са наляво от x са не по-големи от x, а тези надясно от x са не по-малки от него. Така първоначалната задача за сортиране на един масив се свежда до задача за сортиране на два масива (наляво от x и надясно от x). За всеки от тях прилагаме същата функция, т.е. функцията quick е рекурсивна.

```
void quick (struct array *p, int left, int right)
{int i = left, j = right, x = p[ (left +
right)/2].k, y;
  float z;
   do {
     while ((p+i)->k < x  && i < right) i++;
     while ((p+j)->k > x  && j > left) j--;
     if (i<=j)
      { y = (p+i)->k;
        z = (p+i)->x;
         (p+i)->k = (p+j)->k;
         (p+i)->x = (p+j)->x;
         (p+j)->k = y;
         (p+j)->x = z;
         i++; j--;
       }
     }
   while (i<=j);
```

```
  if (left < j) quick (p, left, j);
  if (i < right) quick (p, i, right);
}
```

5. Дефиниране на имена на типове

Една удобна езикова конструкция в C е дефинирането на собствени имена (за програмиста) за типове на променливи. Това се извършва с описанието typedef по следния начин:
typedef стар_тип нов_тип;
където стар_тип е име на допустим за езика тип на променлива, например int, char и т.н.; освен това може стар_тип да е име на тип, дефинирано вече в програмата с typedef;
нов_тип е ново име, заместващо името, описано в стар_тип;
нов_тип е произволен, допустим за езика идентификатор; за да се отличава от останалите идентификатори, обикновено се записва с главни букви; идентификаторът трябва да е различен от имената на другите променливи в съответната област на действие.
Името, определено с typedef, е еквивалентно на името на съществуващ тип, т.е. може да се използва за описание на променливи.

Например описанията:

.....

typedef int PRIM;

.....

PRIM i, j; //дефиниране на променливи i, j от тип PRIM ⇔ тип int;

.....

определят, че променливите i, j са от тип PRIM, който чрез typedef е направен еквивалентен на стандартния тип int.

Описанието typedef може да се използва и със сложни типове данни като масиви, указатели, структури и обединения. В тези случаи се използва следния синтаксис:

```
typedef float EXS[100];
typedef char *WORD;
typedef char *FUN();
typedef struct test
{int k;
 char s;
} DEC;
```

Еквивалентността на новите имена се разбира, както следва:

- EXS може да се използва за описание на масиви със 100 елемента от тип float;

EXS a, b; замества float a[100], b[100];

- WORD може да се използва за описание на указатели към обекти от тип char;

WORD p, q; замества char *p, *q;

- FUN може да се използва за описание на функции, които връщат като резултат указател към обект от тип char;

FUN arc; замества char *arc();

- DEC може да се използва за описание на структурни променливи от тип test;

DEC x; замества struct test x;

Променливите, описани чрез typedef, се използват в програмата така, както е допустимо за съответния тип. Например те могат да участват в изрази, към тях може да се прилага операцията за размер на обект sizeof и т.н.

Необходимо е да се подчертае, че с typedef не се определят нови типове, а само нови имена за съществуващи в езика типове на променливи.
Предимствата при използването на typedef са следните:
- увеличава се прегледността на програмата, тъй като се използват имена, отразяващи конкретното предназначение на променливите;
- при декларациите на променливите се използват по-компактни записи;
- постига се по-добра преносимост на програмите.

Също така, можем да задаваме ново име на тип структура.

```
struct address
 { char town[20];
  char street[20];
  int num;
 }
typedef struct
 { char name[30];
   struct address adr;
   char egn[11];
 } officer;
officer a; //дефиниране на структурна променлива от тип officer
```

По този начин избягваме ключовата дума struct.

6. Полета от битове

В езика C и C++ е възможно да се дефинират набори от последователни битове. Такива набори се наричат полета от битове (разредни полета) и с тях като цяло могат да се осъществяват редица операции. Полетата от битове осигуряват връзката между високото ниво на езика и ниското ниво на апаратната част. Полетата от битове се дефинират като елементи на структура, като се използва следният синтаксис:
unsigned int име_на_полето : размер_на_полето;
където име_на_полето е идентификатор; той се съставя по общите правила и задължително е от тип unsigned int, signed int или int;
размер_на_полето – определя броя на битовете в полето; може да бъде от 0 до 32 включително.

Следният пример илюстрира дефинирането на разредни полета:

```
struct field
 { unsigned int f1:1;
   unsigned int f2:3;
            int k;
 } x,y;
```

Променливите x и y са определени като структури от тип field. Структура от този тип се състои от три елемента: две битови полета f1 и f2 и променливата k от тип int. Разредното поле f1 е от 1 бит, а полето f2 – от 3 бита. Битовите полета са определени в стандарта на езика без специални ограничения относно типа на името и размера им. На практика тези въпроси са решени конкретно за всеки компилатор. Ето защо разредните полета са машинно зависима езикова конструкция.

При определянето на една променлива като структура компилаторът отделя памет за нея, като разполага елементите й последователно.

Ако структурата съдържа разредни полета, те започват да се разполагат в една и съща машинна дума последователно по реда на появяването им. Ако за някое разредно поле няма достатъчно място в текущата дума, то се записва от началото на следващата дума. При това в предишната дума могат да останат неизползвани битове. Ако две разредни полета в тялото на структурата са разделени от описанието на друг елемент, те се записват в различни машинни думи. Например за структурата

```
struct prim
{unsigned int field1:1;
 int k;
 unsigned int field2:5;
} x;
```

полетата field1 и field2 ще се запишат в отделни думи. Обръщението към разредни полета и използването им в изрази е като обикновени елементи на структура – използват се операциите”.” и ”->”.

Например за разгледаната структура от тип prim може да се запише:

```
x.field1 = 0;
x.field2 = 3;
```

или

```
……..
int n;
n = 5 + x.field2/2;
```

Допуска се дефинирането на разредно поле без име. То се използва за отместване в рамките на една машинна дума. Ако размерът на неименуваното разредно поле е 0, непосредствено следващото разредно поле се разполага от началото на следващата машинна дума.

Например:

```
struct bfield
 { unsigned int fd1:1;
                    :0;
   unsigned int fd2:3;
 } bf;
```

Ако подравняването е по байт, тогава имаме следната конфигурация:

| | | | | | | |x| | | | | |x|x|x|

В първия байт се записва полето fd1 (например от младши към старши битове), полето с ширина 0 указва, че другото битово поле fd2 ще се запише в началото на следващия байт.

В стандарта на C и C++ няма специални ограничения относно типа на полето и размера. Тези въпроси са решени отделно за всеки компилатор. Битовите полета са машинно зависима конструкция; най-често използваните типове са unsigned int и int. Размерът на всяко поле не може да надвишава броя на битовете, които се отделят за съответния тип.

Полетата се разполагат от младши към старши битове, но в други реализации разполагането е от старши към младши битове.
Ако на битово поле се присвои величина, която е по-голяма отколкото може да се запише в него, в полето се записват толкова битове, колкото е неговата ширина. При това се губят старшите битове на величината.
Примерна програма:

```
#include <iostream.h>
void main ()
{ struct example
   { int i:2;
     unsigned j:2;
     int :2;
     int k:2;
     int d:8;
   } mystruct;
 mystruct.d = 0;
 mystruct.i = 1;
 mystruct.j = 3;
 mystruct.k = -1;
cout << “Стойността на полетата: ” << mystruct.i << “ “ << mystruct.j
<< “ “ <<  mystruct.k << endl;
}
```

Ако подравняването е по дума имаме следната конфигурация:

|d|d|d|d|d|d|d|d|k|k| | |j|j|i|i|

При полетата signed, най-старшия бит се приема като знаков и затова в битовете за k ще се запише |1|1| и ще се интерпретира като –1.
Използването на битови полета е свързано със следните ограничения:

- не се допускат указатели към битови полета, т.е. към тях не може да се приложи операция &;
- елементите на масив не могат да бъдат битови полета.

7. Обединения

В езика C и C++ е предвидена възможност за описание на област от паметта, в която могат да се записват в различно време променливи от различен тип. Описания от този тип се наричат обединения. Те започват задължително с ключовата дума union и имат следния синтаксис:

```
union име_на_обединението
 { описание_на_елемент1;
   описание_на_елемент2;
   ...
   описание_на_елементn;
 };
```

Има аналогия с описанието на структура. Всички правила за дефиниране и работа със структури са в сила и за обединения. Името на обединението например позволява да се използват дефиниции от вида:

```
union set x;
```

Тази дефиниция определя, че променливата x е обединение с име (тип) set .
Съществената разлика между структури и обединения е паметта, която се резервира при дефинирането на конкретна променлива. За променливите от тип обединение компилаторът разпределя памет, достатъчна за записването на полето с максимална дължина. В така разпределената памет за обединението в даден момент може да се записва само един от елементите на обединението.
Нека разгледаме следното описание:

```
union set
 { int k; //4 байта
   float f; //4 байта
   char c;//1 байт
   char string[5];//5 байта
  } TES;
```

Чрез него се дефинира, че променливата TES е обединение от тип set.

Паметта, която се отделя за TES е равна на паметта за запис на елемента string. Дължината на това обединение е 5 байта. Обръщението към елементите на обединение е същото както при структури – чрез операцията точка или чрез указателната операция, например

```
TES.k = 2;
TES.c = ‘A’;
```

или ако p е указател към обединение от тип set

```
……..
union set *p; //в C++ ключовата дума union не е задължителна
……..
p = &TES;
p->k = 2;
p->c = ‘A’;
union set x, *p = &x;
x.k = 2;
x.f = 7.8; //по този начин губим горната стойност 2
x.c = ‘A’; //по този начин губим горната стойност 7.8
```

С указателна операция:
p->k = 2;
p->f = 7.8;
x->c = ‘A’;
Кой елемент от обединението фактически е записан в паметта се контролира единствено от програмиста. Например в практиката се въвежда допълнителна променлива от цял тип, която да подскаже на програмиста стойност за кое поле е записана в паметта за обединението.
Повечето компилатори поддържат структури с елементи обединения, обединения с елементи структури, масив, чиито елементи са обединения.

По този начин можем да избегнем ограничението елементите на масива да са от един и същи тип, например:

```
union first
 { int k;
   float f;
   char ch;
 } array[10];
```

Елементите на масива array са обединения от тип first и затова в тях могат да се записват променливи от тип int, float или char. Обединението позволява по различен начин да се обръщаме към една и съща област от паметта. Друг случай за използване на обединение – по-особено преобразуване на данните.

x.f = 7.8; Искаме да видим какво е записано в третия байт на обединението – използваме масива string, по-точно string[3].

Обединение може да се инициализира, но само по първия елемент, например:

union set x = { 5};//първия елемент на обединението set е int k; .

8. Изброен тип

Съществуват променливи, които приемат стойности от крайно, предварително зададено множество на целите числа. Типичен пример са булевите променливи, които могат да приемат само две стойности – истина (TRUE) и лъжа (FALSE). В настоящите версии на компилаторите на C и C++ (изброеният тип се въвежда в ANSI C, има го и в C++) са включени данни с изброими стойности, които се описват по следния начин:
enum [име_на_типа] {списък_на_ стойностите} [променливи];
където enum е задължителна ключова дума;
име_на_типа – име на типа на променливата с изброими стойности; може да е всеки допустим за езика идентификатор;
списък_на_ стойностите – последователност от допустими за езика идентификатори, отделени със запетаи;
променливи – идентификатори на променливи, които се дефинират и които могат да приемат стойности само от списъка.

Подобно на структурите можем да дефинираме променливи заедно с описанието на изброения тип.
Да разгледаме следния пример:
enum Boolean { false, true} f;
С това описание променливата f е дефинирана като елемент от данни с изброими стойности. Нейният тип е Boolean и тя може да приеме само две стойности – false и true. Името на типа може да се използва за следващи дефиниции:
enum Boolean i, j; //в C++ enum може да се изпусне
Променливите i и j се описват като данни с изброими стойности от тип Boolean, т.е. могат да приемат само стойности true и false. Идентификаторите в списъка на стойностите се интерпретират като целочислени константи и като такива могат да участват в изрази и оператори. Например следните конструкции са допустими :

```
f = false;
if (i!=true)
 cout << “Грешка в изчисленията!\n”;
```

Компилаторът присвоява на идентификаторите в списъка на стойностите цели числа в нарастващ ред, като започва от 0. Присвояването в списъка се извършва отляво надясно. Така, че f = false; е всъщност f = 0;, но първият оператор прави програмата много по-прегледна и информативна.
Някои компилатори в такъв случай дават съобщение за грешка при преобразуване. Чрез явно преобразуване можем да избегнем съобщението:
f = (Boolean) 0;
Съвременните компилатори позволяват на идентификаторите от списъка на стойностите да се присвояват цели числа по правило, различно от посоченото. Това се извършва при дефинирането на променливите:
enum counter { start, first, second, tenth = 10, eleventh} c;
Компилаторът присвоява на първия идентификатор start 0, на first 1, на second 2, на tenth 10 и на eleventh 11.

Идентификатор в списъка може да бъде инициализиран, като следващите го идентификатори получават нарастващи с 1 стойности, освен ако не са инициализирани.
Броят байтове, които се разпределят за променлива от изброен тип, зависи от стойностите на изброените константи. Например, ако всички константи са от тип char, за променливата ще се разпредели 1 байт; в противен случай се използват 4 байта и стойността на променливата от изброен тип ще се интерпретира като int.
Ако позицията на идентификатор в списъка на стойностите е определена чрез величина от тип int, стойността на идентификатора може да се получи с операцията (тип):

```
.....................
enum EXP { a, b, c, d, e, f} BOX;
int k;
……..……
k = 3;
BOX = (enum EXP)(k+1); ⇔ BOX = e;
```

С този оператор на променливата BOX от тип EXP се присвоява стойността на петия идентификатор в списъка на стойностите, т.е. е.

BOX = k+2;//компилаторът дава съобщение за грешка при преобразуване

Областта на действие на описанието enum е функцията, в която е дефинирано. За да бъде глобално, то трябва да се разположи в началото на файла.

Името на изброения тип, както имената на типове структури и обединения, може да съвпада с имена на променливи в програмата.

```
typedef enum { Monday = 1, Tuesday, Wednesday, Thursday, Friday,
Saturday, Sunday} DAYS;
DAYS week_day, *ptr;
int current_day, to_end, from_start;
cin >> current_day;
week_day = Sunday; //⇔ week_day = (DAYS) 7;
to_end = week_day – current_day;
from_start = Monday;
```

Вход и изход на данни. Текстови и двоични файлове

Входно-изходните операции и операциите за работа с файлове са от изключителна важност във всички процедурни езици за програмиране. В езика С не са предвидени оператори за вход и изход, които по правило са машинно зависими. Вместо тях се използват функции за вход и изход, разпространявани като стандартна библиотека. Тъй като те са стандартизирани, за програмиста е достатъчно да ги познава като имена и начин на използване, без да се интересува от конкретната им реализация за отделните компютри. Общ принцип на входно-изходните функции в езика С е, че с тях се четат и записват байтове, т.е. те са символно ориентирани. Библиотеката е изградена на основата на модела на входно-изходния поток. Входните и изходните данни се организират в поток, представляващ подредена последователност от байтове.

Потокът може да представлява двоични данни или да се разглежда като последователност от ASCII символи. Когато потокът се прехвърля от програмата във файл (или обратно) като двоични данни без преобразувания, се говори за неформатиран двоичен вход-изход. Когато прехвърлянето на данни се извършва съгласно зададен формат, се казва, че има форматиран вход-изход. Например, ако на екрана трябва да се изведе стойността на променлива, която в програмата е обявена като float, в операцията за извеждане, явно или неявно, трябва да има информация за формата на прехвърлянето. В случая на машинно представяне на float трябва да се премине към десетично дробно число, отговарящо на това представяне.
В реалните програми, обработващи данни, е необходимо да се записват и четат данни във и от файлове, записани на диск – дискови файлове. Без тази възможност всички данни, които са въведени или изчислени, се съхраняват само докато е включен компютърът. След изключването му тези данни изчезват.

Има два основни начина за достъп до дискови файлове – буфериран и небуфериран. При буферирания достъп се чете или записва символ по символ и програмистът не се интересува от системно зависими особености, като размери на буфери, сектори и т.н. Често този достъп се нарича вход-изход на високо ниво, тъй като съответните функции осигуряват автоматично необходимите им буфери. При небуферирания достъп всички буфери, броячи, указатели и т.н. са грижа на програмиста, поради което този достъп се нарича вход-изход на ниско ниво или вход-изход от тип UNIX.

Текстовият файл (поток) е последователност от символи. Той може да бъде разделен на редове, като всеки ред е с произволна дължина и завършва със символа за нов ред (‘\n’). Всеки символ се съхранява в отделен байт.

Двоичният файл (поток) е последователност от байтове. Той съдържа информацията във вида, в който тя се съхранява в оперативната памет.

При двоичния файл няма преобразуване на данните – само се прави копие, докато при текстовия файл се прави преобразуване на данните преди се запишат (прочетат). С двоичния файл се прави икономия на външна памет и се печели време. Основно двоичните файлове се използват за съхраняване на данни от тип структура.
Обикновено входът и изходът са буферирани. Буферът е временна област в паметта, в която се запомнят въвежданите и извежданите данни. При извеждане, когато буферът се запълни, неговото съдържание се предава в блок и процеса на буферизация започва отново. Така се ускорява процесът на предаване на данните.
Например при въвеждане функцията getchar() чете един символ от буфера след като в него е записан целия въвеждан ред от клавиатурата (т.е. след натискане на Enter). Функциите getch() и getche(), с прототипи описани в conio.h, четат директно от клавиатурата. Разликата между двете функции е, че getch() не извежда символа на екрана – няма ”exo” на монитора.

При вход и изход от високо ниво, буферите са разположени в областта за динамични данни на потребителската програма.
При вход и изход от ниско ниво, буферите са разположени в работни области на оперативната памет.
Всеки път, когато започва изпълнението на една програма на C, автоматично се отварят три стандартни файла:
stdin – стандартен файл за вход от клавиатурата;
stdout – стандартен файл за изход на екрана;
stderr – стандартен файл за грешки (на екрана).
Тези стандартни файлове са достъпни чрез указатели, които имат същите имена. По подразбиране при scanf се чете от stdin и при printf се извежда в stdout.
Функциите printf и scanf са по-сложни функции за форматиран вход и изход. Такъв вход/изход дава възможност за форматиране (подреждане) на входно-изходната информация.
Функция **printf**
Прототип: int printf(const char **format* [, *argument*]*...*);

Функцията printf форматира и отпечатва (извежда) последователност от символи и стойности към стандартния изходен поток stdout. Ако има аргументи след низа format, низът format трябва да съдържа спецификации, които определят изходния формат за аргументите.

Функция **scanf**

Прототип: int scanf(const char *format [,argument]...);

Функцията scanf чете данни от стандартния входен поток stdin и записва данните в променливите, зададени от argument. Всеки argument трябва да бъде указател към променлива от тип, който съответства на спецификатора за тип във format.

В библиотеката iostream.h са дефинирани предварително четири обекта, свързани с клавиатурата, монитора и устройството за извеждане на съобщения. Обектът cin е от клас istream и е свързан със стандартния вход – клавиатурата. За този вход като файл е определен файлов дескриптор 0. Обектът cout е от клас ostream и е свързан със стандартния изход – монитора. За този изход като файл е определен файлов дескриптор 2.

Обектът cerr е от клас ostream и е свързан със стандартното устройство за извеждане на съобщения (файлов дескриптор 2). По подразбиране това е мониторът. Извеждането чрез този изход не е буферирано. Обектът clog е от клас ostream и е свързан със стандартното устройство за извеждане на съобщения (файлов дескриптор 2). Извеждането чрез този изход е буферирано.
Прието е описаните стандартни обекти да се наричат потоци. За тях в библиотеката iostream.h са дефинирани две операции: >> - за вход и << - за изход. Операнди на операциите са стандартните обекти и данните, които се обработват.
Някои операционни системи позволяват други периферни устройства да се свързват със стандартните файлове – това се указва на ниво операционна система.
Етапи при работа с файлове:

- отваряне на файла
- обработка
  - четене (компонентите се четат последователно от файла без да се променят)

- писане (компонентите се записват във файла в реда на тяхното извеждане)
- актуализация (компонентите се четат от файла и се променя тяхното съдържание)

- затваряне на файла.

9. Текстови файлове. Функции fopen, fclose, getc, putc, fgetc, fputc, ungetc, fscanf, fprintf

Входът и изходът в C се осъществяват със стандартни функции, описани в stdio.h.
Функциите са описани в stdio.h и са от високо ниво за работа с файлове: при функциите от високо ниво, програмистът не се грижи за създаването на небходимите буфери, за тяхния размер и т.н. За работа с файлове се дефинира променлива-указател от тип FILE.
Например:
FILE *fp;

Типът FILE е дефиниран в stdio.h. Това е тип структура, т.е. fp е указател към структура от тип FILE (казваме, че fp е указател към файл). Тази структура включва информация за текущата позиция във файла, указатели към свързаните с него буфери и индикатори за грешки или за достигане на край на файла.

Функция **fopen**

Прототип: FILE *fopen (const char *filename, const char *mode);

Функцията fopen отваря файла с име filename и като резултат връща указател към отворения файл. filename е име на файл, така както той се задава в операционната система, т.е. fopen прави връзка между името на файла и указателя към файла, който се използва по-късно в програмата.

С параметъра mode се определя как ще се използва този файл;

- “r” – отваря файл за четене. Ако файлът не съществува или не може да бъде намерен, се връща NULL.
- “w” – отваря празен файл за писане. Ако даденият файл съществува, съдържанието му се унищожава.

- “a” – отваря за писане в края на файла (добавяне) без изтриване на маркера EOF (End Of File) (-1) преди записване на нови данни към файла. Ако файлът не съществува, първо го създава.
- “r+” – отваря за четене и писане. Файлът трябва да съществува.
- “w+” – отваря празен файл за четене и писане. Ако даденият файл съществува, съдържанието му се унищожава.
- “a+” – отваря за четене и добавяне. Добавянето включва премахването на маркера EOF преди новите данни да се запишат във файла и маркерът EOF се възстановява след като записването приключи. Ако файлът не съществува, първо го създава.

За работа с текстови файлове, към mode прибавяме ‘t’; например: “rt”, “wt”, “r+t” ⇔ “rt+” и т.н. CTRL+Z се интерпретира като символ за край на файла при вход.

За работа с двоични файлове, към mode прибавяме ‘b’; например: “rb”, “ab”, “a+b” ⇔ “ab+” и т.н.

Функцията fopen връща указател NULL при грешка. Ако указател към файл има стойност NULL, това значи, че файлът не е отворен. Например:

- няма място на диска;
- писане върху защитена за писане дискета или диск;
- отваряне на несъществуващ файл за четене.

Когато файлът е отворен за актуализация се позволява въвеждане и извеждане. Важна особеност в режима на актуализация е, че съответните операции се осъществяват спрямо текущата позиция във файла, т.е. ако е въведен даден запис или елемент за актуализация, преди извеждането му указателят в буфера на файла трябва да се позиционира в началото на този запис или елемент с функцията fseek.

Например:

```
FILE *fp;
if ( ( fp = fopen (“name”, “w”))==NULL)
 cout << “Невъзможно е отварянето на файла!\n”;
```

Във всички функции по-надолу се предполага, че указателят fp е инициализиран чрез fopen (с изключение на случаите когато използваме стандартните файлове).

Функция **fclose**

Прототип: int fclose (FILE *fp);

С помощта на тази функция се затваря файл, отворен за буфериран достъп чрез fopen. При това се освобождават всички използвани системни ресурси – буфери, указатели и т.н. По-важно е, че всички данни, които все още не са записани се записват върху външен носител. Не бива да се забравят отворени файлове, тъй като това може да причини загуба на данни, унищожаване на файлове и др.

Ако изпълнението на програмата завършва в главната функция или с функцията exit, автоматично се изпълнява fclose за всички отворени файлове. Въпреки това препоръчително е програмистът да затваря всички използвани файлове.

Функцията връща като резултат EOF (-1), когато има грешка и 0, ако е изпълнена успешно.

Функция **getc**
Прототип: int getc (FILE *fp);
Тази функция чете (въвежда) символа от текущата позиция и увеличава указателят към текущата позиция да сочи към следващия символ от входния поток (файлът сочен от fp). Прочетеният символ се преобразува към int без да се отчита при разширението знаковият бит, т.е. символът се интерпретира като unsigned char и се добавят нули при преобразуването. При успешно изпълнение getc връща прочетения символ. При грешка при въвеждането или при достигане на край на файла, функцията връща EOF.
getc(stdin) ⇔ getchar()
Например:

```
int c;
while ( (c = getc (fp))!=EOF)
 …
```

Функция **putc**
Прототип: int putc (int *c*, FILE *fp);

Функцията извежда (записва) символът c в изходния поток, т.е. във файла, сочен от fp. Тя връща изведения символ или EOF при грешка.
putc (c, stdout) ⇔ putchar()
Например:

```
char msg[] = “Здравейте!\n”
int i = 0;
while (msg[i])
 putc (msg[i++], stdout);
```

Ако функциите getc и putc са реализирани като макроси, тогава в стандартната библиотека на компилатора има функции fgetc и fputc, които не са макроси и са еквивалентни на getc и putc.
Техните прототипи са:

```
int fgetc (FILE *fp);
int fputc (int c, FILE *fp);
```

Функция **ungetc**
Прототип: int ungetc (int c, FILE *fp);

Тази функция връща обратно символа c (например въведен чрез getc или getchar) на съответното място във входния поток (сочен от fp). Тя се използва когато въвеждаме избирателно само определени символи от файла. Функцията връща като резултат върнатия символ при успешно изпълнение или EOF при грешка. Например: въвеждат се цифри и се формира числото, което е зададено с тези цифри.

```
int i = 0, ch;
while ( ( (ch = getchar())!=EOF) && isdigit(ch))
 i = i*10 + ch – ‘0’;
if (ch!=EOF)
  ungetc (ch, stdin); //връщаме символа в буфера на stdin
```

int isdigit (int ch) е макрос, дефиниран в ctype.h, връща 0, ако ch не е цифра и число различно от 0, ако ch е цифра.
Функцията ungetc не е приложима при небуфериран вход.
Функция **fscanf**
Прототип: int fscanf (FILE *fp, const char *format, **...**);

Тази функция има еквивалентно изпълнение на функцията scanf, но чете от произволен входен поток (сочен от fp).
fscanf (stdin) ⇔ scanf
Функция **fprintf**
Прототип: int fprintf (FILE *fp, const char *format, **...**);
Тази функция има еквивалентно изпълнение на функцията printf, но извежда в произволен изходен поток (сочен от fp).
fprintf (stdout) ⇔ printf

10. Текстови файлове. Функции sscanf, sprintf, gets, puts, fgets, fputs

Функция **sscanf**
Прототип: int sscanf (const char *ss, const char *format, **...**);
Функцията sscanf има еквивалентно изпълнение на scanf, но чете от низа ss, записан в оперативната памет. Използва се за преобразуване на низ от символи към числови стойности.

Нулевият байт в края на низа, сочен от ss, се интерпретира като EOF.

Функция **sprintf**

Прототип: int sprintf (char *ss, const char *format, **...**);

Функцията sprintf има еквивалентно изпълнение на printf, но записва в низа ss в оперативната памет. Използва се за преобразуване на числови стойности към символен низ. В края на този низ автоматично се добавя нулев байт.

Функция **gets**

Прототип: char *gets (char *str);

Тази функция се използва за прочитане на цял ред от stdin. Тя преобразува символа за край на ред ‘\n’ в нулев байт. Прочетеният ред ще се прехвърли в низа, сочен от указателя str. Резултатът е указателят към прехвърления низ или NULL при грешка или прочитане на EOF.

Функция **puts**

Прототип: int puts (const char *str);

Пази функция извежда на stdout низа, който се сочи от str. puts преобразува символа за край на низ (‘\0’) в символ за нов ред (‘\n’). При успешно изпълнение puts връща неотрицателно цяло число, при грешка връща EOF.

Функция **fgets**

Прототип: char *fgets (char *str, int n, FILE *fp);

Функцията чете низ от входния поток (сочен от fp) и го прехвърля в низа, сочен от указателя str. Функцията спира въвеждане след прочитане на символа за нов ред, който се съхранява, или след прочитане на n-1 символа или след прочитане на EOF. И в трите случая функцията автоматично добавя нулев байт. При успешно изпълнение, функцията връща указател към прочетения низ, при достигане на EOF или при грешка по време на изпълнение, fgets връща NULL.

Функция **fputs**

Прототип: int fputs (const char *str, FILE *fp);

Функцията fputs извежда във файл, сочен от fp, низът сочен от str.

За разлика от puts, тя не добавя символ за нов ред към извеждания низ. Функцията връща като резултат неотрицателно цяло число при успешно изпълнение или EOF при грешка.
Примерна програма:

```
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h> //функция exit
#define MAX 81
void main ()
 { char name[MAX], address[MAX], ch;
   int age; FILE *fp;
   fp = fopen (“c:\\my.txt”, “w”);
   if (fp==NULL)
    { cout << “Файлът не може да бъде отворен!” << endl;
      exit (1);
    }
   cout << “Въведете данни или Ctrl+Z за край” << endl;
   cout <<  “Име: “ << endl;
```

```
 while ( fgets (name, MAX-1, stdin) != NULL)
  { fputs (name, fp);
    cout << “Адрес: “ << endl;
    fgets (address, MAX-1, stdin);
    fputs (address, fp);
    cout << “Възраст: “ << endl;
    cin >> age;
    fflush (stdin); //ще четем низ след това
    fprintf (fp, “%d”, age);
    fputc (‘\n’, fp); fputc (‘\n’, fp);
    cout << “Име: “ << endl;
  }
fclose (fp);
fp = fopen (“c:\\my.txt”, “r”);
if (fp==NULL)
 { cout << “Файлът не може да бъде отворен!” << endl;
   exit (1);
 }
```

```
  cout << “Съдържание на файла: “ << endl;
  while ( fgets (name, MAX-1, fp) != NULL)
   { fputs (name, stdout);
     fgets (address, MAX-1, fp);
     fputs (address, stdout);
     fscanf (fp, “%d”, &age);
     cout <<  age << endl;
     ch = fgetc (fp); putchar (ch);
     ch = fgetc (fp); putchar (ch);
   }
 fclose (fp);
}
```

11. Двоични файлове. Функции ftell, fseek, rewind, fflush, fread, fwrite

Функция **ftell**
Прототип: long ftell (FILE *fp);
Функцията връща текущата позиция за достъп до данните във файла, сочен от fp. В структурата FILE има поле, в което се съхранява тази текуща позиция. Тя има стойност 0, когато указателят към текущата позиция сочи към началото на файла. Резултатът от функцията при успешно изпълнение е текущата позиция във файла, при грешка функцията връща –1.
Функция **fseek**
Прототип: int fseek (FILE *fp, long offset, int origin);
С тази функция се задава нова текуща позиция за достъп до данните във файла, сочен от fp. За целта се задава offset – адресно отместване в брой байтове и origin – типа на адресното отместване. Той може да бъде:

- началото на файла – тип 0, може да се задава и чрез макрос SEEK_SET;
- текущата позиция – тип 1, може да се задава и чрез макрос SEEK_CUR;
- края на файла – тип 2, може да се задава и чрез макрос SEEK_END.

Функцията връща 0 при успешно изпълнение и цяло число различно от 0 при грешка.

Функция **rewind**

Прототип: void rewind (FILE *fp);

Тази функция позиционира текущата позиция във файла, сочен от fp, в началото на файла. Изпълнението на rewind е еквивалентно на fseek (fp, 0L, SEEK_SET) с тази разлика, че rewind изчиства в структурата FILE индикаторите за край на файл и за грешка, докато fseek изчиства само индикаторът за край на файл.

Функция **fflush**

Прототип: int fflush (FILE *fp);

Тази функция изчиства буферите на файла, сочен от fp. Ако буферите са свързани с изходен файл или с файл за актуализация, то съдържанието им се извежда в този файл. Функцията връща 0 при успешно изпълнение и EOF при грешка.
Функция **fread** – функция за въвеждане без преобразуване
Прототип: size_t fread (void *ptr, size_t size, size_t n, FILE *fp);
Функцията чете от файла, сочен от fp, указаният брой блокове от данни и записва полученото в област от оперативната памет, която се сочи от указателя ptr. Обикновено ptr е указател към структура, size е размер на един блок в брой байтове, n е броят на блоковете за прочитане, общият брой байтове за четене е n*size. При успешно изпълнение функцията връща броя на действително прочетените блокове (n), при грешка или достигане до края на файла връща 0 или цяло число < n.
Функция **fwrite** – функция за извеждане без преобразуване
Прототип: size_t fwrite (void *ptr, size_t size, size_t n, FILE *fp);

Функцията извежда във файла, сочен от fp, указаният брой блокове от данни. ptr е указател към областта от оперативната памет, където са записани данните. Обикновено ptr е указател към структура, size е размер на един блок в брой байтове, n е броят на блоковете за извеждане, общият брой байтове за извеждане е n*size. При успешно изпълнение функцията връща броя на действително изведените блокове (n), при грешка или достигане до края на файла връща 0 или цяло число <n.

Важно: функциите ftell, fseek се отнасят за двоични файлове. При текстови файлове има ограничения (поради преобразуването) – например за функцията fseek аргументът offset трябва да е 0 или стойност, която функцията ftell е върнала при предишно извикване за същия файл, също така аргументът origin трябва да има стойност SEEK_SET (0).

Примерна програма: създава двоичен файл с компоненти структури от данни за студенти – име, група, среден успех и извежда студентът с най-висок среден успех.

```
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h> //функция exit
#include <conio.h> //функция getche
#include <ctype.h> //функция toupper
struct student
 { char name[60];
   int group;
   float av_score;
 } ;
int writefile (char *);
void readfile (char *);
void main ()
 { int i; char name_file[30];
   cout << “Име на файл: “ << endl;
   gets (name_file);
   i = writefile (name_file);
   if (i!=EOF)
```

```
   cout << “Файлът е създаден!” << endl;
  else {
   cout << “Файлът не е създаден.” << endl;
   exit (1);
   }
  readfile (name_file);
 }
 int writefile (char *name_file)
  { FILE *fp;
    struct student s;
    if ( (fp = fopen (name_file, “wb”) ) == NULL)
     { cout << “Файл с име “ << name_file << ” не може да се създаде”
<< endl;
       exit (1);
     }
do {
      cout << “Име: “ << endl; gets (s.name);
      cout << “Група: “ << endl; cin >> s.group;
```

```
      cout << “Среден успех: “ << endl; cin >> s.av_score;
      fflush (stdin);
      fwrite (&s, sizeof (s), 1, fp);
      cout << “Ще въвеждате ли още? (Y/N): “<< endl;
    } while ( toupper (getche ()) == ‘Y’ );
   return fclose (fp);
  }
void readfile (char *name_file)
{ FILE *fp;
  struct student s;
  int flag = 0;
  long int pos;
  float max_av;
  if ( (fp = fopen (name_file, “rb”)) == NULL )
   { cout << “Файлът не може да бъде отворен!” << endl;
     exit (1);
   }
  while ( fread (&s, sizeof (s), 1, fp) == 1)
```

```
   { if (!flag)
      { max_av = s.av_score; pos = 0; flag = 1; }
    else if (s.av_score > max_av)
     { max_av = s.av_score;
       pos = ftell (fp) – sizeof (s);
     }
   }
  fseek (fp, pos, SEEK_SET);
  fread (&s, sizeof (s), 1, fp);
  cout <<  s.name << “ “ << s.group << “ “ << s.av_score << endl;
  fclose (fp);
}
```

12. Структурно и модулно програмиране. Същност на обектно-ориентираното програмиране

Практиката на програмирането е показала, че когато се съставят програми, най-трудното е откриването на грешки. Една от първите причини за това е прекомерното използване на оператор goto в програмите. Пръв Е. Дийкстра през 1965 г. защитава тезата срещу оператора goto и се счита, че той полага принципите на структурното програмиране. Негова е мисълта, че ”операторът GOTO може да бъде изключен от езиците за програмиране”. Основната идея на структурното програмиране е ограничаването на допустимите управляващи структури. В теоретични работи е доказано, че са достатъчни само два вида. Според теорията, за да се напише една програма са необходими точно три основни компоненти: две управляващи и една ”изпълнителна”.

При структурното програмиране има три структури за управление:

- последователно изпълнение (например в езика С операторите се изпълняват последователно);
- селектиране (в С оператора if е основно средство);
- повторение, изключва се goto (в С основно средство е цикъл while).

Структурното програмиране е един подход за създаване на ясни, добре документирани и структурирани програми.

Принципите на структурното програмиране са:

- принцип на визуалност и мнемоника (краснопис в програмирането);
- принцип за програмиране без goto;
- принцип за стандартизация на управлението;
- програмата да се проектира предварително, преди да се кодира;
- програмата да съдържа коментари;
- разбиване на една програма на подпрограми (функции) – стъпка по стъпка да се детайлизира алгоритъма.

Подходът на структурното програмиране все още е неефективен при по-големи програми. Усъвършенства се използването на подпрограмите – в структурното програмиране не са дефинирани ясно връзките между отделните подпрограми. В резултат на развитието на подпрограмите се появява модул (например езика MODULA 2). Основна особеност на модулното програмиране е стандартизацията на интерфейса между отделните програмни единици. Модулът е отделна функционално завършена програмна единица, която структурно се оформя по стандартен начин по отношение на компилатора и по отношение на обединяването й с другите аналогични единици и зареждането за изпълнение. Като правило, всеки модул съдържа описание, в което са указани всичките му основни характеристики: език за програмиране, обем, входни и изходни променливи, техния формат, ограничения върху тях и т.н. Обемът на модула обикновено не надвишава 1000 оператори на езика за програмиране. В противен случай модулът става тромав и труден за възприемане и използване.

Модулното програмиране е начин на разбиване на задачата на известен брой различни модули, умение да се използват стандартни модули чрез параметрична настройка, автоматизация на сглобяването на готови модули от библиотеки, банки от модули.

Основни концепции на модулното програмиране:

- всеки модул реализира една независима функция;
- всеки модул има една точка за вход и една точка за изход;
- размерът на модула по възможност трябва да е минимизиран;
- всеки модул може да бъде разработен и кодиран от различни членове на бригадата програмисти и може да бъде отделно тестван;
- цялата система е построена от модули;
- модулът не трябва да има странични ефекти;
- всеки модул не зависи от това, как са реализирани другите модули.

При такъв подход сложната система се разделя на няколко части, едновременно създавани от различни програмисти.

Всеки модул реализира единствена функция. Размерът на модула не е голям, затова тестването е управляемо и може да бъде проведено най-щателно. След кодирането и тестването на всички модули те се интегрират и се тества цялата система.

Въвеждането на понятието модул позволява още по-големи програми.

По-нататък модулът се развива към понятието **клас**, т.е. модулното програмиране се развива към **обектно-ориентирано програмиране.** Новото е, че структурата данни, около която се създава класа е извлечена от естествените характеристики на моделирания обект. Обектно-ориентираните езици по-добре представят семантиката на сложния обект. Обектно-ориентираното програмиране капсулира данни (атрибути) и функции (варианти на поведение) в съвкупности, наречени **обекти.** Данните и функциите на обекта са тясно свързани помежду си. Обектите притежават свойството да скриват информацията.

Това означава, че макар обектите да знаят как да се свързват един с друг чрез добре определен интерфейс, те обикновено не знаят как са реализирани другите обекти – детайлите на реализацията са скрити вътре в самите обекти.
При структурното програмиране в C и C++, основна програмна единица е функцията. При обектно-ориентираното програмиране това е класът. Под полиморфизъм се разбира възможността обекти от различни класове, свързани с помощта на наследяване, да реагират по различен начин при обръщение към една и съща функция-елемент.

13. Класове и обекти. Примерна програма

В C и другите процедурно-ориентирани езици програмирането е ориентирано към действието, докато програмирането на C++ е обектно-ориентирано.

В С единица за програмиране е функцията. В С++ единица за програмиране е класът, на основата на който в крайна сметка се създават обектите.
Основно вниманието на програмистите на С е съсредоточено в създаването на функции. Групата действия, изпълняващи някаква задача, се обединява във функция, а функциите се групират, за да образуват програма. Данните несъмнено са важни в С, но съвременната гледна точка е, че данните съществуват за поддържане на действията, изпълнявани от функциите. Глаголите в описанието на проектираната система помагат на програмиста на С да определи множеството от функции, които съвместно ще работят за реализация на системата.
Основно вниманието на програмистите на С++ е съсредоточено в създаването на свои собствени, определяни от потребителя типове, наречени класове. Класовете са типове, дефинирани (въведени) от програмиста. Всеки клас съдържа данни и набор от функции, обработващи тези данни.

Данните, които са компоненти на класа, се наричат данни-елементи (или данни-членове), функциите, които са компоненти на класа, се наричат функции-елементи (или функции-членове). Подобно на това, както екземпляр от вграден тип, например int, се нарича променлива, екземпляр от потребителски дефиниран тип (клас) се нарича **обект**. Програмистът използва вградените типове като блокове за конструиране на определените от потребителя типове. Център на вниманието в C++ са не функциите, а обектите. Съществителните имена в описанието на проектираната система помагат на програмиста на C++ да определи множеството от класове. Тези класове се използват за създаване на обектите, които ще работят съвместно за реализацията на системата. Класовете в C++ са естествено продължение на структурата struct в C.
Класовете дават възможност да се моделират обекти, които имат атрибути, представени като данни-елементи, и варианти на поведение, представени като функции-елементи.

Типовете, съдържащи данни-елементи и функции-елементи, обикновено се определят в C++ с ключовата дума class. Понякога функциите-елементи в езиците за обектно-ориентирано програмиране се наричат **методи**, те се извикват в отговор на съобщения, изпратени към обекта. Съобщението съответства на извикването на функция-елемент.
Когато класът е определен, името на класа може да се използва за дефиниране на обект от този клас. Ще дефинираме клас Time. Декларацията на класа Time започва с ключовата дума class. Тялото за деклариране на клас се загражда в големи скоби { }. Декларацията на клас завършва с точка и запетая. Декларацията на класа Time съдържа три цели числа като елементи: hour, minute и second.
Синтаксис на декларация на клас:

```
class име_на_клас
 {
   тяло_на_клас
 } ;
```

Примерна декларация на клас:

```
class Time {
 public:
  Time();
  void setTime (int, int, int);
  void printMilitary();
  void printStandard();
 private:
  int hour; // 0 - 23
  int minute; // 0 - 59
  int second; // 0 - 59
} ;
```

Етикетите public: (открити) и private: (скрити) се наричат **спецификатори за достъп** до елементите.

Данните-елементи и функциите-елементи декларирани след public: до следващия спецификатор за достъп се наричат **открити** и те са достъпни при всяко обръщение на програмата към обект от класа Time. Наричат се още глобални.
Данните-елементи и функциите-елементи, декларирани след private: до следващия спецификатор за достъп, се наричат **скрити** и те са достъпни само за функциите-елементи от този клас. Наричат се още локални.
Спецификаторите за достъп завършват с ':', могат да се появяват многократно в декларацията и в произволен ред. Препоръчва се, обаче, в декларацията на клас всеки спецификатор за достъп до елемент да се използва само веднъж, започвайки с public.
В примера функциите-елементи са открити, те представляват открит интерфейс на класа и ще се използват за обработка на данните от този клас.
Функцията-елемент Time има същото име като на класа Time.

Тя се нарича **конструктор** на този клас – това е специална функция-елемент, която инициализира данните-елементи на обект от този клас. Конструкторът на класа се извиква автоматично при създаване на обект от този клас. Обикновено класът има няколко конструктори - това се постига чрез предефиниране на функции.
За конструктора не се определя тип.
Целите данни-елементи hour, minute, second са достъпни само за функциите-елементи на класа.
Обикновено данните-елементите са в частта private, а функциите-елементи в частта public.
След като класът е деклариран, името му може да се използва като тип за дефиниране на обекти от този клас.
Например:
Time s, a[5], *p, &d = s; // обект, масив от обекти, указател към обект, псевдоним на обект от тип Time

Примерна програма:

```
#include <iostream.h>
class Time {
 public:
  Time();  // конструктор
  void setTime (int, int, int);  // задава времето
  void printMilitary();  // печат на времето във военен формат
  void printStandard();  // печат на времето във стандартен формат
 private:
  int hour;
  int minute;
  int second;
} ;
Time::Time()
 { hour = minute = second = 0; }
```

void Time::setTime (int h, int m, int s) // задаване на нова стойност на времето във военен формат. Проверка за правилност. Нулиране на неправилни стойности.

```
 { hour = (h >=0 && h < 24) ? h : 0;
   minute = (m>=0 && m < 60) ? m : 0;
   second = (s>=0 && s < 60) ? s : 0;
  }
void Time::printMilitary()  // печат на времето във военен формат
 { cout << (hour<10 ? “0” : “”) << hour << “:”
       << (minute<10 ? “0” : “”) << minute << “:”
       << (second<10 ? “0” : “”) << second << endl; }
void Time::printStandard ()  // печат на времето във стандартен
формат
 { cout << ( (hour==0 || hour ==12) ? 12 : hour % 12) << “:”
       << (minute<10 ? “0” : “”) << minute << “:”
       << (second<10 ? “0” : “”) << second << ((hour < 12) ? “ AM” : “
PM”)
            << endl;
  }
```

```
void main ()
{ Time t; // при това дефиниране автоматично се извиква
конструктора на класа Time, създава се нов екземпляр t на обект
от класа Time
cout << “Начална стойност на времето във военен формат ”;
  t.printMilitary();//резултат 00:00:00
cout << “\n Начална стойност на времето в стандартен формат”;
  t.printStandard();//резултат 12:00:00 AM
  t.setTime (13, 27, 6);
cout << “\n\nВремето във военен формат след setTime”;
  t.printMilitary();//резултат 13:27:06
cout << “\n\nСтандартно време след setTime”;
  t.printStandard();//резултат 1:27:06 PM
  t.setTime (99, 99, 99); // задаване на неправилни стойности
cout << “\n\nСлед опита за неправилно задаване:” << ”\nВоенно
време: ”;
  t.printMilitary();//резултат 00:00:00
cout << “\nСтандартно време: ”;
```

```
  t.printStandard();//резултат 12:00:00 AM
cout << endl;
}
```

C :: се означава двуместна операция за разрешаване на област на действие. Тъй като различните класове могат да имат елементи с еднакви имена, операцията за разрешаване на област на действие привързва името на елемента към името на съответния клас. Въпреки че функция-елемент е декларирана в декларацията на класа, тя може да бъде описана извън тази декларация, тя и в този случай има област на действие клас. Ако функция-елемент е описана в декларацията на класа, тя автоматично се разглежда като вградена, т.е. inline. Функция-елемент, описана извън декларацията на класа, може да бъде направена вградена чрез явно използване на ключовата дума inline. Ще отбележим, че компилаторът може да не вгражда никакви функции. Описанието на неголеми функции-елементи в декларацията на класа подобрява производителността, ако компилаторът ги вгражда, но не подобрява качеството на техниката на програмиране.

cout е стандартен обект, който е свързан със стандартния поток за извеждане, който най-често операционната система свързва с екрана. endl е манипулатор, можем вместо него да изведем ‘\n’ – разликата е, че манипулаторът изчиства буфера за извеждане, т.е. предизвиква незабавно извеждане на всички данни в буфера.
Функциите-елементи printMilitary и printStandard нямат аргументи. Това е защото функциите-елементи неявно ”знаят”, че те трябва да отпечатат данни-елементи на определения обект от тип Time, за който те са активирани. Това прави извикванията на функциите-елементи по-кратки от съответните извиквания на функции в процедурното програмиране.
Ще отбележим, че данните-елементи не могат да получават начални стойности в декларацията на класа, където са дефинирани. Тези данни-елементи трябва да бъдат инициализирани чрез конструктора на класа или чрез другите функции (set).

Обикновено функциите-елементи имат по-малко параметри, тъй като те директно могат да използват данните-елементи в съответния клас. Това намалява вероятността за грешки при обръщенията към тях.
Функция със същото име, както на класа, но започваща с ‘~’ се нарича **деструктор** на този клас. Деструкторът изпълнява завършващи служебни действия за всеки обект на класа, преди паметта, разпределена за този обект, да бъде повторно използвана от системата.
Откритите функции реализират възможностите на класа, необходими на неговите клиенти (т.е. части от програмата, играещи ролята на потребители) – те се наричат **интерфейс** на класа. Декларирането на функциите-елементи в тялото на класа и тяхното описание извън него, отделят интерфейса на класа от неговата реализация – това подпомага разработката на качествено програмно осигуряване.
Клиентите на класа го използват без да знаят вътрешните детайли на неговата реализация.

Ако реализацията на класа се промени, интерфейсът на класа остава непроменен и текстът на програмата на клиента не се променя – това значително опростява модификацията на програмите. Тази тенденция се задълбочава с използването на скрити данни-елементи.

14. Област на действие клас и достъп до елементите на класа

Данните-елементи и функциите-елементи на класа имат **област на действие клас**. Функциите, които не са елементи на клас, имат област на действие файл.
В областта на действие клас елементите на класа са непосредствено достъпни за всички функции-елементи на този клас и те могат да се използват просто по име.

Извън областта на действие клас към елементите на класа може да се обръщаме или с помощта на името на обекта, или чрез псевдоним на обекта, или чрез указател към обекта.
Функциите-елементи на класа могат да се предефинират, но само с помощта на други функции-елементи от този клас. Типична грешка е опитът за предефиниране на функция-елемент с помощта на функция, която не е от областта на действие на този клас.
Компонентите на функциите-елементи имат област на действие функция – например променливите, които са дефинирани в тялото на функция-елемент са известни само на тази функция.
Ако във функция-елемент се дефинира променлива с име, съвпадащо с името на променлива от област на действие клас, то първата скрива втората. Такива скрити променливи могат да станат достъпни чрез бинарната операция за разрешаване на област на действие с ляв операнд името на класа и десен операнд името на променливата.

Скритите глобални променливи могат да станат достъпни с помощта на унарната операция за разрешаване на област на действие.
Например:

```
float value = 1.23456;
void main ()
 { int value = 7;
   cout << value << endl << ::value << endl;
  //тук ::value означава глобалната променлива value;
 }
```

Операциите за достъп до елемент на клас са аналогични на операциите за достъп до елемент на структура:

- операцията '**.**' за достъп до елемент на обект има синтаксис:
  име_на_обект**.**име_на_елемент;
  вместо име_на_обект може да имаме псевдоним на този обект;
- операцията '->' за достъп до елемент на обект има синтаксис:
  указател_към_обект->име_на_елемент.

Пример: прост клас Count с открити данна-елемент x от тип int и открита функция-елемент print за илюстрация на достъпа до елементите на класа с помощта на операциите за избор на елемент. По принцип трябва да се избягват открити данни!

```
#include <iostream.h>
class Count
 { public:
    int x;
    void print () { cout << x << endl; }
 } ;
void main ()
 { Count counter,  // създава се обект counter
   *counterPtr = &counter,  // указател към counter
   &counterRef = counter;  // псевдоним на counter
   cout << “Присвояване на x стойност 7 и отпечатване по името на обекта”;
   counter.x = 7;
   counter.print ();
```

```
   cout << “Присвояване на x стойност 8 и отпечатване по
псевдоним”;
   counterRef.x = 8;
   counterRef.print ();
   cout << “Присвояване на x стойност 10 и отпечатване по
указател”;
   counterPtr->x = 10;
   counterPtr->print ();
 }
```

Спецификаторите за достъп до елемент public, private, protected (защитен) се използват за управление на достъпа до данните-елементи и функциите-елементи на класа. По премълчаване спецификаторът за достъп е private, т.е. всички елементи след заглавието до първия спецификатор за достъп са скрити. Скритите елементи са достъпни само за функциите-елементи и за приятелските функции на този клас. Откритите елементи на класа са достъпни за всички функции в програмата.

Скритите елементи на класа и описанията на откритите функции-елементи са недостъпни за клиентите на класа. Типична грешка е да се прави опит за достъп до скритите елементи на даден клас във функция, която не е елемент на този клас или негов приятел. Пример с класа Time: скритите елементи на класа са достъпни само чрез открития интерфейс на класа, включващ открити функции-елементи.

```
void main ()
 { Time t;
   t.hour = 7;// компилаторът дава грешка – Time::hour е недостъпна
   cout << t.minute;// компилаторът дава грешка – Time::minute е
недостъпна
 }
```

В дефиницията на класа се препоръчва най-напред да се използва спецификатор за достъп public и след това private. Това концентрира вниманието на клиентите на класа върху неговия открит интерфейс.

Добър стил на програмиране е използването на скрити данни-елементи и открити функции-елементи за задаване и получаване на стойностите на скритите данни-елементи.
Клиент на класа може да бъде функция-елемент на друг клас или глобална функция.
Структурите в C++ са разширени с възможности за задаване на режим на достъп до елементите и с възможности за включване на функции-елементи. В структурите по премълчаване се приема достъп public. В C++ структурите и класовете са взаимнозаменяеми.
Създателите на класове използват режимите на достъп за да осигуряват скриване на информацията и прилагане на принципа за най-малките привилегии.
Функциите-елементи или приятелските функции на един клас, които осигуряват достъп до скритите данни-елементи на класа се наричат **функции за достъп**.

Например много често се използва функция за достъп с име get, която е за прочитане на стойностите на скритите данни или функция за достъп с име set за задаване и изменение на скрити данни. Ако е необходимо тези две функции биха могли да извършват проверка за коректността на данните. Те, също така, могат да преобразуват данните от вида, използван в интефейса, във вида, използван при реализацията.
Друго типично използване на функциите за достъп е проверката на определено условие – такива функции се наричат **предикатни функции**. Пример за предикатна функция е функцията isEmpty за всеки клас-контейнер, т.е. клас, който може да съдържа много елементи, например свързан списък, стек, опашка. Програмата ще използва функцията isEmpty преди да се опита да прочете поредния елемент от обекта на контейнера. Предикатната функция isFull би могла да проверява обект от клас-контейнер дали има достатъчно място в него.

Подобни обслужващи функции не са част от интерфейса и те не са предназначени за използване от клиентите на класа, затова те са скрити функции-елементи и се използват за обслужване на другите функции-елементи на класа.
Примерна програма: обслужваща функция с клас SalesPerson с 12 данни за месечните продажби, инициализирани с конструктора. Техните стойности се задават със setSales. Откритата функция printAnnualSales печата сумата на продажбите за последните 12 месеца. Обслужващата функция totalAnnualSales сумира сведенията за продажбите за 12 месеца, осигурявайки работа на printAnnualSales. Функцията-елемент printAnnualSales редактира сведенията за продажбите и ги отпечатва като сума в долари. Функцията main включва само проста последователност от извиквания на функции-елементи без никакви управляващи структури.

```
#include <iostream.h>
#include <iomanip.h>// за параметризираните манипулатори
class SalesPerson
 { public:
     SalesPerson ();
     void SetSales (int, double);
     void getSalesFromUser();
     void PrintAnnualSales ();
   private:
     double Sales[12];
     double TotalAnnualSales ();
  };
SalesPerson::SalesPerson ()
 { for (int i = 0; i < 12; i++)
    Sales[i] = 0;
 }
void SalesPerson::getSalesFromUser()
```

```
{double SalesFigure;
   for (int i = 1; i <=12; i++)
    { cout << “Въведете продажби за месец “ << i << “: “;
      cin >> SalesFigure;
      SetSales (i, SalesFigure);
    }
}
void SalesPerson::SetSales ( int month, double amount)
 { if (month>=1 && month <=12 && amount > 0)
    Sales [month-1] = amount;
   else cout << “Грешка!” << endl;
 }
double SalesPerson::TotalAnnualSales ()
 { double total = 0;
   for (int i = 0; i < 12; i++)
    total += Sales[i];
   return total;
 }
```

```
void SalesPerson::PrintAnnualSales ()
 { cout << setprecision (2)
        << setiosflags (ios::fixed | ios::showpoint)
        << “Сума на продажбите: $“
        << TotalAnnualSales() << endl;
 }
void main ()
 { SalesPerson S;
   S.getSalesFromUser();
   S.PrintAnnualSales();
 }
```

Използвали сме параметризиран манипулатор:
setprecision (int n)
Той изисква включването на заглавния файл iomanip.h. Този манипулатор установява точност от n знака след десетичната точка при извеждане на числа с плаваща точка. По премълчаване точността е 6 знака след десетичната точка. При това се извършва закръгляне към най-близкото цяло число.

Манипулаторът:
setiosflags (long f)
задава форматни спецификации, определени чрез флага f. Константата ios::fixed води до извеждане на числата с плаваща точка като дробно-десетични. Константата ios::showpoint води до извеждане на десетичната точка и нулевите младши разряди. Двете константи са свързани с ‘|’ (побитово или), тъй като и двете трябва да се включат във флага f.

15. Интерфейс и реализация

Отделянето на интерфейса от реализацията е един от най-фундаменталните принципи за създаване на добро програмно осигуряване. Това улеснява модификацията на програмата. Измененията в реализацията на класа не влияят на клиента, ако интерфейсът остане същия.

Препоръчва се дефиницията на класа да се оформи като отделен заглавен файл. Така тя е достъпна за всеки клиент, който иска да използва класа. Това формира открития **интерфейс** на класа. Препоръчва се описанието на функциите-елементи на класа да се оформи в изходния файл. Това формира **реализацията** на класа. Заглавният файл, в който е описана дефиницията на класа, обаче, може да съдържа известна част от реализацията и кратки сведения за други части от реализацията. Вградените функции-елементи, например, трябва да се намират в заглавния файл – по този начин клиентът може да включва определени функции inline. Скритите елементи са изброени в заглавния файл, така че тези елементи са видими за клиента, макар че той няма достъп до тях. Заглавните файлове се включват чрез директивата include във всеки файл, в който се използва класа. Тези първични файлове и първичният файл с главната функция се компилират разделно. Илюстрация (с класа Time)

Заглавен файл time1.h

```
// Деклариране на класа Time
// Функциите-елементи са дефинирани в time1.cpp
#ifndef TIME1_H
#define TIME1_H

Дефиниране на класа Time…

#endif
```

Първичен файл time1.cpp

```
#include “time1.h”
#include <iostream.h>

Описание на функциите елементи...
```

Първичен файл prog.cpp

```
#include “time1.h”
```

#include <iostream.h>

Описание на главната функция...

Тук главната функция се явява като клиент.
Използваните директиви на препроцесора в заглавния файл time1.h предотвратяват включването на кода между #ifndef и #endif, ако е дефиниран идентификатора TIME1_H.
Ако заглавният файл още не е включван в даден първичен файл, то се дефинира идентификатора TIME1_H чрез директивата define и заглавният файл се включва в първичния файл. Ако заглавният файл е вече включен, това означава, че идентификаторът TIME1_H е вече дефиниран и заглавният файл не се включва повторно в първичния файл.
Опити за многократно включване на заглавен файл обикновено се правят в големи програми с много заглавни файлове, които от своя страна включват други заглавни файлове.

Мнемонично е името на макроса да се получава от името на заглавния файл (например чрез замяната на ‘.’ с ‘_’).

16. Конструктори и деструктори

След създаване на един обект, неговите елементи могат да бъдат инициализирани с помощта на функция **конструктор.**
Конструкторът е функция-елемент, която има същото име, както името на класа. Конструкторът се извиква автоматично при създаването на обекта (създаване на екземпляр на класа). За конструкторите не може да се задава тип на връщания резултат и те не връщат резултат.
Конструкторите могат да се предефинират и така да се определи множество от начални стойности за инициализация на обектите на класа. Данните-елементи трябва да се инициализират в конструктор на класа или техните стойности може да се зададат по-късно след създаването на обекта.

Когато се дефинира обект от даден клас, между името на обекта и ‘;’ може в скоби да се зададе списък за инициализация на елементи – тези начални стойности се предават като параметри в конструктора на класа.
Конструктор, при който всички аргументи са по премълчаване или който няма аргументи, се нарича **конструктор по премълчаване.**
За всеки клас може да съществува само един конструктор по премълчаване.
Модификация на класа Time – с конструктор с елементи по премълчаване.

```
#include <iostream.h>
class Time
 { public:
    Time (int = 0, int = 0, int = 0); // конструктор по премълчаване
    void setTime (int, int, int);
    void printMilitary ();
    void printStandard ();
```

```
private:
     int hour;
     int minute;
     int second;
  };
Time::Time (int hr, int min, int sec)
 { setTime (hr, min, sec); }
…
описание на другите функции-елементи
…
void main ()
 { Time t1, // всичките аргументи са по премълчаване
        t2 (2), // минутите и секундите са по премълчаване
        t3 (21, 34), // секундите са по премълчаване
        t4 (12, 25, 42), // всички стойности са зададени
        t5 (27, 74, 99); // всичките стойности са неправилно зададени
   t1.printMilitary ();    // 00:00:00  всичките аргументи са по
премълчаване
```

```
   t2.printMilitary ();    //  02:00:00  минутите и секундите са по
премълчаване
   t3.printMilitary ();    //  21:34:00  секундите са по премълчаване
   t4.printStandard (); // 12:25:42 PM  всички стойности са зададени
   t5.printStandard (); // 12:00:00 AM  всичките стойности са
неправилно зададени
 }
```

Тук конструкторът извиква функцията SetTime със стойности, предадени на конструктора или със стойности по премълчаване. Това извикване улеснява съпровождането на програмата, тъй като при промяна на реализацията на SetTime, това ще бъде отразено и в конструктора.

Въпреки това се губи ефективност – това може да се поправи, ако конструкторът Time явно се обяви като inline. Ако конструкторът Time е описан вътре в дефиницията на класа, той по подразбиране се обявява като inline.

Добър стил на програмиране е декларирането на аргументи по премълчаване само в прототипа на функцията-елемент вътре в дефиницията на класа в заглавния файл. Грешка е задаването на начални стойности по премълчаване за една и съща функция-елемент както в прототипа, така и в нейното описание.
Ако за класа не е дефиниран конструктор, то компилаторът създава конструктор по премълчаване. Този конструктор не задава начални стойности, така че след създаването на обекта, данните в него могат да са некоректни.
Деструкторът е специална функция-елемент на класа. Името на деструктора съвпада с името на класа, но пред него се поставя символ тилда ‘~’. Мнемониката е, че ‘~’ означава поразрядно логическо отрицание.
И така деструкторът може да се разглежда като отрицание на конструктора. Деструкторът на класа се извиква при унищожаване на обекта. Например, деструкторът за даден обект се извиква когато изпълнението на програмата напусне областта на действие на този обект.

В действителност, самият деструктор не унищожава обекта, той извършва финализиращи действия, преди системата да освободи паметта, в която е бил съхраняван обекта за да се използва за създаване на нови обекти. Деструкторът няма параметри и не връща резултат. Не се разрешава предефиниране на деструктора, т.е. един клас може да има само един деструктор.
На практика деструкторите рядко се използват в обикновени класове. Деструкторите имат смисъл за класове, които използват динамично разпределение на паметта за обектите.
Конструкторите и деструкторите се извикват автоматично. Редът, в който се изпълняват тези извиквания, зависи от реда, в който изпълнението на програмата влиза или излиза от областите на действие на обектите. В общия случай, извикването на деструкторите се изпълнява в ред, обратен на извикването на конструкторите на съответните обекти.

Класът памет на обектите може да измени този ред:

- конструкторите на обекти, дефинирани като глобални се извикват преди изпълнението на функциите от даден файл (включително и main). Съответните деструктори се извикват когато завърши изпълнението на главната функция main или когато се извика функция exit;
- конструкторите на автоматичните локални обекти се извикват, когато изпълнението на програмата достигне мястото, където те са дефинирани. Съответните деструктори се изпълняват когато се напусне областта на действие на обектите, т.е. когато се напусне блока, в който тези обекти са дефинирани. Конструкторите и деструкторите на автоматичните обекти се извикват всеки път при влизане или излизане от областта им на действие. Деструкторите не се извикват за автоматични обекти ако изпълнението на програмата завърши с извикването на функцията exit или abort;
- конструкторите на статичните локални обекти се извикват веднага, когато изпълнението на програмата достигне за първи път мястото, където те са дефинирани.

Съответните деструктори се извикват когато завършва главната функция main или когато се извиква функция exit. Деструкторите не се извикват за обекти static ако изпълнението на програмата завършва с извикване на функцията abort.
Примерна програма, илюстрираща реда на извикване на конструкторите и деструкторите. Програмата декларира глобален обект first. Неговият конструктор се извиква веднага щом програмата започва да се изпълнява, а неговият деструктор се извиква след завършването на програмата, след като всички други обект са унищожени.
Функцията main декларира три обекта. Обектите second и fourth са локални автоматични обекти, а обектът third е статичен локален обект. Конструкторите на всеки от тези обекти се извикват когато процесът на изпълнение достига мястото, където обектите са декларирани. Деструкторите на обектите fourth и second се извикват в съответния ред, когато завършва main.

Понеже обектът third е статичен, той съществува до завършването на програмата. Деструкторът на обекта third се извиква по-рано от деструктора на first, но след унищожаването на всички други обекти.
Функцията Create декларира три обекта — локалните автоматични обекти fifth и seventh и статичния локален обект sixth.
Деструкторите за обектите seventh и fifth се извикват в съответния ред след завършването на Create. Тъй като sixth е статичен обект, той съществува до завършването на програмата. Деструкторът за sixth се извиква преди деструктора за third и first, но след унищожаването на всички други обекти.

```
#include <iostream.h>
class CreateAndDestroy
 { public:
    CreateAndDestroy (int); // конструктор
    ~CreateAndDestroy ();  // деструктор
```

```
  private
    int data;
} ;
CreateAndDestroy::CreateAndDestroy (int value)
 { data = value; cout << “Обект “ << data << “ конструктор”; }
CreateAndDestroy::~CreateAndDestroy()
 { cout << “Обект “ << data << “ деструктор ” << endl; }
void Create ();
CreateAndDestroy first (1); // глобален обект
void main ()
 { cout << “ глобален създаден преди main” << endl;
   CreateAndDestroy second (2); // локален обект
   cout << “локален автоматичен в main” << endl;
   static CreateAndDestroy third (3); // локален обект
   cout << “локален статичен в main” << endl;
   Create (); // извикване на функцията за създаване на обекти
   CreateAndDestroy fourth (4); // локален обект
   cout << “локален автоматичен в main” << endl;
```

```
 }
void Create ()
 { CreateAndDestroy fifth (5);
   cout << “локален автоматичен в Create” << endl;
   static CreateAndDestroy sixth (6);
   cout << “локален статичен в Create” << endl;
   CreateAndDestroy seventh (7);
   cout << “локален автоматичен в Create” << endl;
 }
```

Резултат от изпълнението:

```
Обект 1 конструктор глобален създаден преди main
Обект 2 конструктор локален автоматичен в main
Обект 3 конструктор локален статичен в main
Обект 5 конструктор локален автоматичен в Create
Обект 6 конструктор локален статичен в Create
Обект 7 конструктор локален автоматичен в Create
Обект 7 деструктор
```

Обект 5 деструктор
Обект 4 конструктор локален автоматичен в main
Обект 4 деструктор
Обект 2 деструктор
Обект 6 деструктор
Обект 3 деструктор
Обект 1 деструктор

17. Използване на данни-елементи и функции-елементи

Скритите данни-елементи на един клас могат да се променят само с помощта на функции-елементи или приятелски функции на този клас. Тези функции трябва да проверяват коректността на данните.

Класовете обикновено съдържат открити функции-елементи, чрез които клиентите на класа задават (записват) или получават (четат) стойностите на скритите данни-елементи.
Ако данните-елементи са открити, те могат директно да бъдат записвани и четени от всяка функция в програмата. Използването на открити данни-елементи, обаче, не се препоръчва.
Пример с класа Time, който го разширява с включване на функции за четене и запис на скрити данни-елементи hour, minute и second. Функции за запис стабилно управляват задаването на данните-елементи. Опитите да се зададат на данни-елементи неправилни стойности предизвикват присвояване на тези данни-елементи на нулеви стойности (и по такъв начин привеждане на данните-елементи в непротиворечиво състояние). Всяка функция за четене просто връща съответната стойност на данните-елементи.
Програмата отначало използва функция за запис да зададе правилни стойности на скритите данни-елементи на обекта t на класа Time, след това използва функция за четене да изведе тези стойности на екрана.

Функцията incrementMinutes не е елемент на класа. Затова тя използва функции-елементи за запис и четене за съответното увеличение на елемента minute. Това функционира правилно, но снижава производителността поради многократните извиквания на функциите. По-нататък ще обсъдим дружествените функции като средство за отстраняване на този недостатък.

заглавен файл time.h

```
#ifndef TIME_H
#define TIME_H
class Time
 { public:
    Time (int = 0, int = 0, int = 0);
    void setTime (int, int, int); // задаване на час, минути, секунди
    void setHour (int); // задаване на час
    void setMinute (int); // задаване на минути
    void setSecond (int); // задаване на секунди
    int getHour (); // получаване на час
```

```
    int getMinute ();// получаване на минути
    int getSecond ();// получаване на секунди
    void printMilitary ();
    void printStandard ();
   private:
  int hour;
  int minute;
  int second;
};
#endif
```

<u>първичен файл time.cpp</u>

```
#include <iostream.h>
#include “time.h”
…
описание на функциите Time, setTime, printMilitary, printStandard
…
```

```
void Time::setHour (int h)
 { hour = (h>=0 && h < 24) ? h : 0; }
void Time::setMinute (int m)
 { minute = (m>=0 && m < 60) ? m : 0; }
void Time::setSecond (int s)
 { second = (s>=0 && s < 60) ? s : 0; }
int Time::getHour () { return hour; }
int Time::getMinute () { return minute; }
int Time::getSecond () { return second; }
```

първичен файл maintime.cpp

```
#include <iostream.h>
#include “time.h”
void incrementMinutes (Time &, const int);
void main ()
 { Time t;
   t.setHour (17); t.setMinute (34); t.setSecond (25);
```

```
  cout << “Час: “ << t.getHour () << endl
       << “Минути: “ << t.getMinute () << endl
       << “Секунди: “ << t.getSecond () << endl;
  t.setHour (234); t.setMinute (43); t.setSecond (6373);
  // hour = 0, minute = 43, second = 0
  t.setTime (11, 58, 0);
  incrementMinutes (t, 3);
  t.printMilitary ();
}
void incrementMinutes (Time &tt, const int count)
{ cout << “Увеличаване на минутите с ” << count << “\nНачално
време: ”;
   tt.printStandard();
   for (int i = 0; i<count; i++)
   { tt.setMinute ( (tt.getMinute () + 1) % 60);
     if (tt.getMinute () == 0)
      tt.setHour ( (tt.getHour() + 1) % 24);
     cout << “\nминути + 1: ”;
```

```
    tt.printStandard();}
cout << endl;
 }
```

В случая формалният параметър е псевдоним на фактическия параметър. Достъпът до скритите данни-елементи се осъществява с функциите get и set, което води до намаляване на ефективността. Този недостатък може да се отстрани чрез използване на приятелски функции.

Възможно е чрез открита функция-елемент на клас да се върне в програмата неконстантен псевдоним на скрита данна-елемент от този клас. Пример за връщане на псевдоним на скрита данна-елемент. Програмата започва с обявяването на обекта t от класа Time и псевдонима hourRef, на който е присвоена стойност, връщана от извикването на t.badSetHour(20). Програмата извежда на екрана стойността на псевдонима hourRef. По-нататък този псевдоним се използва за задаване на стойност на hour равна на 30 (неправилна стойност) и тази стойност отново се извежда на екрана. Накрая като L-величина се използва извикване на самата

функция, на нея се присвоява стойност 74 (друга неправилна стойност), която също се извежда на екрана.

```
#include <iostream.h>
class Time {
  public:
     Time (int = 0, int = 0, int = 0);
     void setTime (int, int, int);
     int getHour ();
     int& badsetHour (int);
  private:
     int hour, int minute, int second;
} ;
…
описание на Time и на setTime
…
int Time::getHour ()
 { return hour; }
int &Time::badsetHour (int h)
```

```
 { hour = (h >=0 && h < 24) ? h : 0;
   return hour;//връща псевдоним на скритата данна елемент hour
 } ;
void main ()
 { Time t;
   int &hourRef = t.badsetHour (20); // hourRef става псевдоним на
скритата данна- //елемент hour
   cout << “Часове преди задаването: ” << hourRef;
   hourRef = 30; //не се прави проверка за коректност на данните
cout << “Часове след задаването: ”<< t.getHour () << endl;
   t.badsetHour (12) = 74; //аналогично
 }
```

Препоръчва се открита функция-елемент да не връща неконстантен указател или псевдоним на скрита данна-елемент, тъй като това нарушава инкапсулацията на класа.

Един обект може да се присвои (=) на обект от същия клас.

Такова присвояване обикновено се извършва чрез побитово копиране, т.е. всеки елемент на единия обект се копира индивидуално в съответния елемент на другия обект. Побитовото копиране може да предизвика сериозни проблеми, когато се прилага за класове, чиито данни-елементи използват динамично разпределение на паметта.
Обектите могат да се предават като параметри на функции и могат да се връщат като резултат. По премълчаване се предава или връща копие на обекта, т.е. се реализира предаване по стойност. Обектът също може да бъде предаден или върнат чрез псевдоним или указател. За да не се променя предавания или връщания обект, може да се използва ключовата дума const.
Съществуват много библиотеки от класове, други са в процес на разработка. Целта е тези библиотеки да станат широкодостъпни за да може при създаване на програмно осигуряване да се използват добре описани и внимателно проверени компоненти.

Проблем е да се установи дали съществува клас, който съответства на конкретни изисквания на разработчиците на програмно осигуряване.

18. Константни обекти и константни елементи

Константните обекти и константните елементи са едни от средствата за прилагане на един от най-фундаменталните принципи за създаване на добро програмно осигуряване – принципът за най-малките привилегии. Ключовата дума const се използва за да се укаже, че обектът е неизменяем, т.е. константен, и че всеки опит да се промени обекта е грешка.
Пример:
const Time noon (12, 0, 0); // дефиниране на константен обект от клас Time
Опитите да се промени константен обект се откриват като грешка от компилатора.

Не може да се извикат функции-елементи на константни обекти ако самите функции-елементи също не са обявени като const. Това важи и за get функциите, които не променят обекта. Функциите-елементи, обявени като const, не могат да променят обект – компилаторът не позволява това.
Една функция-елемент се обявява като константна, като и в дефиницията на класа, и в описанието на функцията се поставя ключовата дума const след заглавието на функцията, но преди лявата голяма скоба на тялото на функцията. Например:
int A::getvalue() const {return privateDateMember;};
За конструкторите и деструкторите не се указва const, тъй като конструкторът трябва да има възможност да даде начални стойности на данните-елементи на обекта, а деструкторът трябва да има възможност да извършва финализиращи действия преди унищожаването на обекта.
Препоръчва се да се описват като константни всички функции-елементи, за които се предполага, че ще се използват с константни обекти.

Типична грешка е да се описва като константна функция-елемент, която извиква неконстантна функция-елемент на класа.
Константна функция-елемент може да бъде предефинирана с неконстантен вариант. Изборът коя от функциите-елементи да се използва, се извършва от компилатора автоматично, в зависимост от това дали обекта, който извиква функцията е константен или не.
Пример (разширение на класа Time):

```
#include <iostream.h>
class Time
 { public:
    Time (int = 0, int = 0, int = 0);
    void setTime (int, int, int);
    void setHour (int);
    void setMinute (int);
    void setSecond (int);
    int getHour () const;
    int getMinute () const;
    int getSecond () const;
```

```
    void printMilitary () const;
    void printStandard ();
  private:
    int hour;
    int minute;
    int second;
 }
…
описание на функциите Time, setTime, setHour, setMinute,
setSecond
…
int Time::getHour () const { return hour; }
int Time::getMinute () const { return minute; }
int Time::getSecond () const { return second; }
void Time::printMilitary () const
 { …
   описание на функция printMilitary
   …
```

```
}
void Time::printStandard ()
{ …
  описание на функция printStandard
  …
}
void main ()
{ const Time t (19, 33, 52);
  t.setHour (12); //грешка
  t.setMinute (20); //грешка
  t.setSecond (39); //грешка
}
```

В горния пример Borland C++ дава само предупреждение, че неконстантна функция-елемент се използва с константен обект. Това означава, че самата функция фактически ще се изпълни, ако се пренебрегне предупреждението (константният обект ще се промени без съобщение за грешка). Константни функции-елементи могат да се използват без предупреждение с константни обекти.

В Microsoft Visual C++ това не са предупредителни грешки, а синтактични грешки.
Ако данни-елементи на класа са дефинирани като константни (const), тогава трябва да се използват инициализатори на елементи, за да може конструкторът на този клас да зададе начални стойности на тези данни-елементи.
Пример: използване на инициализатора на елементи на класа за задаване на начална стойност на константния елемент increment на класа Increment. Конструкторът за Increment се изменя по следния начин:

```
Increment :: Increment  (int c, int i)
          : increment (i)
    { count = c; } `
```

Записът : increment (i) предизвиква задаване на начална стойност на елемента increment, равна на i.

```
#include <iostream.h>
class Increment
 { public:
```

```
    Increment (int = 0, int = 1);
    void addIncrement () { count += increment; }
    void print () const;
   private:
    int count;
    const int increment;
  } ;
 Increment::Increment (int c, int i) : increment (i)
 //използвали сме инициализатор, за да дадем начална стойност
на
// константната данна елемент;
  { count = c; }
 void Increment::print () const
  { cout << “count = “ << count
         << “increment = “ << increment
         << endl;
  }
 void main ()
```

```
{ Increment value (10, 5);
  cout << "Преди увеличението: ";
  value.print ();
  for (int i = 0; i <3; i++)
   { value.addIncrement ();
     cout << "След увеличението “ << i +1 << “: “;
     value.print ();
   }
}
```

Ако е необходимо да се дадат начални стойности на няколко константни данни-елементи, те се изреждат в списък, разделени със запетаи. Чрез инициализатори могат да се присвояват стойности и на неконстантни данни-елементи.

Ако направим опит да инициализираме константен елемент чрез директно присвояване, а не чрез инициализатор, компилаторът дава предупреждение – неинициализиран константен елемент и съобщение за грешка – константен обект не може да се променя с присвояване.

19. Композиция – класове като елементи на други класове (влагане на класове)

Един клас може да включва в себе си обекти от други класове като данни-елементи.
Както е известно, когато обект от даден клас се дефинира, автоматично се извиква неговият конструктор. Ако класът съдържа обекти-елементи, в неговия конструктор трябва да се укаже как аргументите се предават към конструкторите на обектите-елементи. Обектите-елементи се създават в този ред, в който са декларирани, преди да бъдат създадени обектите на включващия ги клас.
Начални стойности за обектите-елементи могат да се задават чрез списък от инициализатори към конструктора на включващия ги клас. Ако не е зададен списък от инициализатори, за обектите-елементи автоматично се извиква конструктор с аргументи по премълчаване.

След това те могат да бъдат променяни чрез съответните функции-елементи за запис (set). Типична грешка е да не е предвиден конструктор с аргументи по премълчаване за обекти-елементи, за които не е зададен инициализатор, в резултат съответният обект-елемент може да остане неинициализиран (въпреки, че за него е извикан служебен конструктор по премълчаване). От съображения за ефективност се препоръчва обектите-елементи да се инициалират явно, с помощта на инициализатори.
Примерна програма: тя използва класовете Employee и Date за демонстрация на обекти като елементи на други обекти. Класът Employee съдържа скрити данни-елементи lastName, firstName, birthDate и hireDate. Елементите birthDate и hireDate са обекти от класа Date, който съдържа скрити данни-елементи month, day и year. Програмата създава обекта Employee, задава началните стойности на неговите данни-елементи и ги показва на екрана. Синтаксисът на прототипа на функцията в описанието на конструктора Employee е:

```
Employee::Employee (char *fname, char *lname,
                    int bday, int bmonth, int byear,
                    int hday, int hmonth, int hyear)
      : birthDate(bday, bmonth, byear),
        hireDay(hday, hmonth, hyear)
```

Този конструктор има осем аргумента (fname, lname, bday, bmonth, byear, hday, hmonth и hyear). Двоеточието в прототипа отделя инициализаторите на елементите от списъка на параметрите. Инициализаторите на елементите указват, че аргументите на Employee се предават на конструкторите на обектите-елементи. В частност, bday, bmonth и byear се предават на конструктора birthDate, а hday, hmonth и hyear на конструктора hireDay. Инициализаторите на елементите в списъка се разделят със запетаи.

```
#include <iostream.h>
#include <string.h>
class Date
 { public:
```

```
    Date (int = 1, int = 1, int = 1900);
    void print () const;
    ~Date();
  private:
    int month; int day; int year;
    int checkDay (int);
} ;
Date::Date (int d, int m, int y)
 { if (m > 0 && m <=12) month = m;
   else { month = 1;
          cout << “Месец “ << m << “ е некоректен! Зададен е месец
1!”
                  << endl;
        }
   year = y;
   day = checkDay (d);
   cout << “Конструктор на обекта Date “;
   print (); cout << endl;
```

```
 }
int Date::checkDay (int testday)
 { static const int dayspermonth [13] = { 0, 31, 28, 31, 30, 31, 30, 31,
31, 30,
                                    31, 30, 31} ;
   if (testday > 0 && testday <= dayspermonth [month])
    { return testday; }
   if (month == 2 && testday == 29 && ( year % 400 == 0 ||
      ( year % 4 == 0 && year % 100 != 0))) return testday;
   cout << “Денят “ << testday << “ е некоректен! Зададен е ден 1!”
           << endl;
   return 1;
 }
void Date::print () const
 { cout << day << ‘/’ << month << ‘/’ << year; }
Date::~Date ()
{ cout << “Деструктор на Date обект за дата ”;
  print ();
```

```
  cout << endl;
}

class Employee
{ public:
   Employee (char *, char *, int, int, int, int, int, int);
   void print() const;
~Employee ();
  private:
    char firstname[25];
    char lastname[25];
    const Date birthdate;
    const Date hiredate;
} ;
Employee::Employee (char *fname, char *lname, int bday, int bmonth,
int byear, int hday, int hmonth, int hyear) : birthDate (bday, bmonth,
byear), hireDay (hday, hmonth, hyear)
//използвали сме инициализатори, за да зададем аргументи за
```

```
//конструкторите на обектите-елементи
{ int length = strlen (fname);
  length = (length < 25) ? length : 24;
  strncpy (firstname, fname, length);
  firstname[length] = '\0';
  length = strlen (lname);
  length = (length < 25) ? length : 24;
  strncpy (lastname, lname, length);
  lastname[length] = '\0';
  cout << "Конструктор на обекта Employee: " << firstname << ' '
       << lastname << endl;
}
void Employee::print () const
 { cout << lastname << ", " << firstname << endl
        << "Постъпил на работа: ";
   hireDate.print ();
   cout << " Роден на: ";
   birthDay.print ();
```

```
   cout << endl;
 };
Employee::~Employee ()
{cout << “Деструктор на обекта Employee: ” << lastName << “, “ <<
firstName << endl;
}

void main ()
 { Employee e (“Иван”, “Петров”, 24, 7, 1949, 12, 3, 1988);
   cout << “\n”;
   e.print ();
   cout <<”\nПроверка на конструктора Date с неправилни
стойности: \n”;
   Date d (35, 14, 94);
   d.print ();
 }
```

20. Приятелски функции и приятелски класове

Приятелските функции на един клас се дефинират извън областта на действие на този клас, но имат право на достъп до скритите елементи (private) и защитените елементи (protected) на дадения клас.
Функция или цял клас могат да бъдат обявени като приятели (friend) на друг клас. Приятелските функции се използват за повишаване на производителността, например при предефиниране на операции или при създаване на класове итератори.
Обектите от клас итератор се използват за да избират последователно елементи или да изпълняват операции над елементите на обект от клас контейнер. Типични операции върху елементи на обект от клас контейнер са вмъкване, изключване, търсене, сортиране, проверка за наличие на елемент в класа и др. Примери за класове контейнери са масиви, стекове, опашки, свързани списъци.

За да се обяви, че функция е приятел на клас, преди прототипа на функцията в описанието на класа се записва ключовата дума friend.
За да се обяви например, че ClassTwo е приятел на ClassOne, трябва в описанието на ClassOne да се запише:
friend ClassTwo;
В този случай всички функции-елементи на ClassTwo стават приятелски функции за ClassOne.
Обикновено класът на итераторите се обявява като приятелски на класа на контейнерите.
Обявяване на приятелство може да стане на произволно място в описанието на класа. Препоръчва се обаче, това да става в началото, преди спецификаторите за достъп до елемент.
Някои програмисти смятат, че приятелството вреди на скриването на информация и отслабва обектно-ориентирания подход.
Примерна програма: деклариране и използване на приятелска функция setx за задаване на стойност на скрит елемент x от

данните от класа Count. Ще отбележим, че декларацията friend е първа в декларацията на класа, преди всичко друго.

```
#include <iostream.h>
class Count
 { friend void setx ( Count &, int);
   public:
     Count () { x = 0;}
     void print () const { cout << x << endl; }
   private:
     int x;
 } ;
void setx ( Count &c, int value)
 { c.x = value; }
void main ()
 { Count counter;
    cout << “counter.x след създаването: ”;
    counter.print ();
```

```
    cout << “counter.x след извикването на приятелската функция
setx: ”;
    setx (counter, 8);
    counter.print ();
 }
```

Ако функцията не беше обявена за приятел на класа Count, този начин на използване на данната-елемент ще доведе до съобщение за грешка.

21. Използване на указателя this

Всеки обект може да определи своя собствен адрес с помощта на указателя this. Указателят this на обекта не е част от самия обект. Указателят this се предава в обекта от компилатора като неявен първи аргумент във всяко извикване на функция-елемент за обекта. Този указател неявно се използва за достъп както до данните-елементи, така и до функциите-елементи на обекта. Също така, той може да се използва и явно.

Указателят this се инициализира автоматично да сочи към обекта, за който се извиква функция-елемент. Типът на указателя this зависи от типа на обекта и от това, обявена ли е функцията-елемент, в която се използва this, за const.
В неконстантна функция-елемент, указателят this има тип тип_на_обекта *const, т.е. указателят е константен.
В константна функция-елемент, указателят this има тип const тип_на_обекта *const, т.е. това е константен указател към обект, който също е константа.
Всяка нестатична функция-елемент има достъп до указателя this на обекта, за който е извикана тази функция-елемент.
С цел икономия на памет, за всяка функция-елемент съществува само едно копие на клас и тази функция-елемент се извиква от всеки обект на дадения клас. От друга страна, всеки обект има собствено копие на данните-елементи на класа.
Пример за използване на this: показано е явно използване на указателя this за да се даде възможност на функцията-елемент на класа Test да печата скрита променлива от обекта Test.

Най-напред функцията-елемент print печата x непосредствено. След това програмата използва два различни записа за достъп към x чрез указателя this – операция стрелка (->), приложена към указателя this, и операция точка (.).

```
#include <iostream.h>
class Test
 { public:
     Test (int = 0);
     void print () const;
   private:
     int x;
 }
Test::Test (int a) { x = a; }
void Test::print () const
 { cout << "        x = " << x << endl
          << " this->x = " << this->x << endl
          << " (*this).x = " << (*this).x « endl;
}
```

```
void main ()
 { Test a (12);
   a.print ();
 }
```

В израза (*this).x малките скоби са задължителни, тъй като операцията ‘.’ има по-голям приоритет от операцията *. Във функцията-елемент print по три начина (които на машинно ниво не се различават) се обръщаме към данната-елемент x на обекта Time.

Едно интересно приложение на указателя this е предотвратяване на присвояване на един обект сам на себе си. При предефиниране на операции, самоприсвояването може да стане причина за сериозни грешки в случаите, когато обектът съдържа указател към динамично разпределена памет.

Друго приложение на указателя this е възможността за слепване на извиквания на функция-елемент.

Примерна програма (с класа Time): илюстрира се връщане на псевдоним на обекта Time, което дава възможност за слепване на извикванията на функции-елементи на класа Time. Всяка от функциите-елементи setTime, set Hour, setMinute и setSecond връща *this с тип на връщане Time &.

```
#include <iostream.h>
class Time
 { public:
    Time (int = 0, int = 0, int = 0);
    Time& setTime (int, int, int);
    Time& setHour (int);
    Time& setMinute (int);
    Time& setSecond (int);
    int getHour () const;
    int getMinute () const;
    int getSecond () const;
    void printMilitary () const;
    void printStandard () const;
```

```
 private:
    int hour;
    int minute;
    int second;
} ;
Time::Time ( int h, int m, int s)
{ setTime (h, m, s); }
Time& Time::setTime (int h, int m, int s)
{ hour = (h>=0 && h < 24) ? h : 0;
  minute = (m>=0 && m < 60) ? m : 0;
  second = (s>=0 && s < 60) ? s : 0;
  return *this;
}
Time& Time::setHour (int h)
{ hour = (h>=0 && h < 24) ? h : 0;
  return *this;
}
Time& Time::setMinute (int m)
```

```
 { minute = (m>=0 && m < 60) ? m : 0;
   return *this;
 }
Time& Time::setSecond (int s)
 { second = (s>=0 && s < 60) ? s : 0;
   return *this;
 }
…
описание на функциите getHour, getMinute, getSecond,
                    printMilitary, printStandard
…
void main ()
 { Time t;
   t.setHour(18).setMinute(30).setSecond(22);
   cout << “Военно време: ”;
   //операцията ‘.’ е ляво-асоциативна
   t.printMilitary();
   cout << “\nСтандартно време: ”;
```

```
  t.printStandard();
  cout << “\n\nНов стандарт на времето: “;
  t.setTime (20, 20, 20).printStandard();
}
```

22. Статични елементи на клас

Обикновено, всеки обект на клас има свое собствено копие на всички данни-елементи на класа. В определени случаи, обаче, е нужно да се използва само едно копие на променлива съвместно от всички обекти на класа. За тези и за други цели се използват статична променлива в класа, която съдържа информация за ползване от всички обекти на целия клас.
Декларирането на статичните данни-елементи започва с ключовата дума static.

Препоръчва се, с цел икономия на памет, ако е достатъчно единствено копие на данните, да се използват статични данни-елементи.
Статичните данни-елементи имат област на действие клас. Те могат да бъдат public, private, protected.
На статичните данни-елементи могат да се задават начални стойности само веднъж в областта на действие файл.
Достъпът до откритите (public) статични елементи на класа е възможен посредством всеки обект на класа или посредством името на класа, с помощта на бинарната операция за разрешаване на област на действие. Вижда се, че статичните данни-елементи на клас са достъпни (съществуват) дори когато не съществуват обекти от този клас. За да се осигури достъп до открит статичен елемент, просто се поставя пред данната-елемент името на класа и бинарна операция за разрешаване на област на действие.

За осигуряване на достъп до скритите и защитените статични елементи трябва да се предвиди открита статична функция-елемент, която се извиква, като се използва бинарната операция за разрешаване на област на действие.
Една функция-елемент може да бъде декларирана като static, ако тя не трябва да има достъп до нестатични елементи на класа. За разлика от нестатичните функции-елементи, в статичните не може да се използва указател this, тъй като статичните данни-елементи и статичните функции-елементи съществуват независимо от обектите на класа. Типична грешка е използването на указателя this в статична функция-елемент. Друга типична грешка е деклариране на статична функция-елемент като const.
Примерна програма: показва се използването на скрита статична данна-елемент и открита статична функция-елемент. Данната-елемент count се инциализира с 0 в областта на действие файл с оператора
int Employee::count = 0;

Данната-елемент count обслужва броенето на елементите на класа Employee, които са създадени. Ако обекти от клас Employee съществуват, елементът count може да бъде извикан посредством всяка функция-елемент на обекта Employee (в дадения пример както с конструктора, так и с деструктора).

```
#include <iostream.h>
#include <assert.h> //използване на обслужваща функция assert
#include <string.h>
class Employee
 { public:
    Employee (const char *, const char *);
    ~Employee ();
    const char *getFirstName () const;
    const char *getLastName () const;
    static int getCount (); //връща броя на създадените обекти
   private:
    char *firstName; char *lastName; static int count;
 } ;
```

```
int Employee::count = 0;
//инициализиране на статичен елемент на клас в областта на
//действие файл
int Employee::getCount () { return count;}
Employee::Employee (const char *first, const char *last)
 { firstName = new char [strlen(first) + 1];
   assert (firstName != NULL);
   strcpy (firstName, first);
   lastName = new char [strlen(last) + 1];
   assert (lastName != NULL);
   strcpy (lastName, last);
   count++;
   cout << “Конструктор Employee за “ << firstName << ‘ ‘
        << lastName << “ е извикан,” << endl;
 }
Employee::~Employee ()
 { cout << “Деструктор Employee е извикан за “ << firstName << ‘ ‘
          << lastName << endl;
```

```
   delete []firstName;
   delete []lastName;
   count--;
 }
const char *Employee::getFirstName () const { return firstName;}
const char *Employee::getLastName () const { return lastName;}
//функциите връщат указатели към константа, за да се
предотврати
//промяната на скритите данни-елементи от клиента;
void main ()
 { cout << “Брой служещи преди създаването на обектите “ <<
Employee::getCount() << endl;
   Employee *e1ptr = new Employee (“Петър”, “Радев”);
   Employee *e2ptr = new Employee (“Иван”, “Михайлов”);
   cout << “Брой служещи след създаването на обектите “ << e1ptr-
>getCount() << endl;
   cout <<“\n\nСлужещ 1: “
          << e1ptr->getFirstName()
```

```
        << ‘ ‘ << e1ptr->getLastName()
        <<“\nСлужещ 2: “
        << e2ptr->getFirstName()
        << ‘ ‘ << e2ptr->getLastName()
        << endl;
  delete e1ptr; // освобождаване на памет
  e1ptr = 0;
  delete e2ptr; // освобождаване на памет
  e2ptr = 0;
  cout << "Брой служещи след освобождаването " <<
Employee::getCount() << endl;
 }
```

Функцията assert е оператор за контрол. Нейният прототип е описан в заглавния файл assert.h. Тя проверява стойността на аргумента, който е израз. Ако стойността на израза е 0, т.е. лъжа, assert извежда съобщение за грешка и извиква функция abort от библиотеката на стандартните функции, която завършва изпълнението на програмата.

Съобщението за грешка съдържа проверяваният израз, името на файла, съдържащ оператора за контрол и номера на реда, на който се намира този оператор.
Прототипът на функцията abort е описан в stdlib.h, но за използването на assert този файл не трябва да се включва.
Операторите за контрол не трябва да се отстраняват от програмата след завършване на нейната проверка. Когато операторите за контрол не са вече нужни, в началото на първичния файл с програмата (преди включване на assert.h) се включва директивата
#define NDEBUG
В резултат на това, препроцесорът игнорира всички оператори за контрол.
Операцията new автоматично разпределя памет за обект от съответния размер, извиква конструктора на обекта и връща указател към него. Ако new не може да намери необходимата памет, тя връща указател NULL.

Да разгледаме следния оператор:
TypeName *typeNamePtr;
TypeName е име на тип.
За да се създаде динамичен обект от тип TypeName, в C++ просто се пише:
typeNamePtr = new TypeName;
Операцията delete освобождава паметта за обект, разпределена преди това с операцията new за същия обект. Също така, операцията delete автоматично извиква деструктора на обекта.
За да се освободи памет, разпределена по-рано с new, в C++ просто се пише:
delete typeNamePtr;
C++ позволява да се използва инициализатор за току-що създаден обект:
float *thingPtr = new float (3.14159);
който задава начална стойност на новосъздадения обект от тип float, равна на 3.14159.

Масив от 10 цели числа може да се създаде и да се присвои на arrayPtr по следния начин:
int *arrayPtr = new int[10];
Този масив може да бъде унищожен с помощта на оператора:
delete [] arrayPtr;

23. Предефиниране на операции

Манипулирането с обектите на класовете се реализира чрез изпращане на съобщение до обектите, т.е. чрез извикване на функции-елементи за обектите. Понякога тези обръщения са дълги и обемисти, особено за математически класове. В тези случаи е удобно да се използват наличните в C++ набори от вградени операции за работа с обекти, което се постига с помощта на механизма за предефиниране на операции.
Например операцията с означение ‘<<’ се използва в C++ за различни цели – като операция за побитово отместване вляво или като операция за извеждане в поток.

По същия начин, операцията ‘>>’ е операция за побитово отместване вдясно или операция за въвеждане от поток. Всяка от тези операции е **предефинирана** в библиотеката от класове на C++.
Друг пример за предефиниране на операции в C++ са операциите ‘+’ и ‘-‘ – те се изпълняват по различен начин при цели операнди, при операнди с плаваща точка или при адресна аритметика.
C++ не позволява да се създават означения за нови операции.
Препоръчва се да се използва предефиниране на операции, само ако това прави програмата по-ясна в сравнение с използването на явни обръщения към функциите, които изпълняват тези операции.
Операциите се предефинират чрез функция с име, което се състои от ключовата дума operator и означението на операцията.
Например operator+ е име на предефиниращa функция за операцията за събиране.
За да се използва операция с обекти на класове, тази операция трябва да бъде предефинирана с две изключения:

1. Операцията за присвояване (=) може да се използва с всеки клас без явно да се предефинира. По премълчаване тази операция се свежда до побитово копиране на данните-елементи на класа. Такова побитово копиране, обаче, е опасно за класове с елементи указатели, които сочат към динамично разпределена памет. В такъв случай след присвояване на един обект на друг, те ще сочат към една и съща динамично разпределена област от паметта. Изпълнението на деструктора на кой да е от тези обекти ще освободи тази област от паметта и ако след това чрез другия обект има обръщение към вече освободената памет, резултатът ще бъде неопределен. За такива класове явно трябва да се предефинира операцията за присвояване;
2. Операцията за адресиране (&) също може да се използва с обекти от всеки клас без да се предефинира. Тя връща адреса на обекта в паметта. Въпреки това тази операция може да се предефинира.

Предефинирането на операции е характерно за математически класове, например класове за комплексни числа. Препоръчва се операциите да се предефинират така, че те да извършват над обектите на класа същата функция, или близка до нея, каквато те изпълняват над обектите от вградените типове.
Отделните компилатори имат особености при работа с предефинирани операции. Повечето операции в C++ могат да бъдат предефинирани. Не могат да бъдат предефинирани операциите за условен израз (?:), за разрешаване на област на действие (::), операцията '.' за избор на елемент на обект и операцията sizeof.
Приоритетът и асоциативността на операциите не могат да бъдат променяни при тяхното предефиниране. При предефинирането на операции не могат да се използват аргументи по премълчаване. Също така не е възможно да се промени броя на операндите на операциите – предефинираните унарни операции остават унарни, а предефинираните бинарни – бинарни.

Операциите ‘+’, ‘*’, ‘&’, ‘-‘ имат бинарен и унарен вариант и тези варианти се предефинират отделно.
Предефинирането не може да промени начина на изпълнение на операциите с вградени типове. Желателно е, поне един от операндите на предефинирана операция да е обект от тип, дефиниран от потребителя.
Предефиниране на присвояването и събирането позволяват да напишем
object2 = object2 + object1;
но това не означава, че операцията += също е предефинирана, за да можем да напишем
object2 += object1;
Това може да се постигне чрез явно предефиниране на операцията += за дадения клас. За да се осигури съгласуваност на такива свързани операции се препоръчва при предефинирането на едните да се използват другите. Например при предефинирането на += да се използва операцията +, която вече е предефинирана.

В различни случаи предефиниращите функции е най-добре да бъдат приятелски функции или функции-елементи или обикновени функции.
Функциите-операции могат да бъдат или не функции-елементи. Ако функциите не са елементи, те обикновено са приятелски.
Функциите-елементи неявно използват указателя this, за да получат един от своите аргументи във вид на обект на класа. Този аргумент трябва да бъде указан явно в списъка при извикването на функция, която не е елемент.
При предефиниране на операциите за извикване на функция ‘()’, за достъп до елемент на масив ‘[]’, указателната операция ‘->’ и операцията за присвояване ‘=’, функцията за предефиниране на операцията трябва да бъде обявена като елемент на класа. За други операции функциите за предефиниране на операции могат да не са функции-елементи (тогава те обикновено са приятелски).
Реализирана ли е функцията-операция като функция-елемент или не, операцията в изразите се реализира еднакво. Коя реализация е по-добра?

Когато функцията-операция е реализирана като функция-елемент, крайният ляв (или единствен) операнд трябва да бъде обект от същия клас (или псевдоним на обект от същия клас), елемент на който е функцията. Ако левият операнд трябва да е обект от друг клас или вграден тип, такава функция-операция не може да бъде реализирана като функция-елемент. Функцията-операция, реализирана не като функция-елемент, трябва да бъде приятелска ако тази функция трябва да има пряк достъп до скритите или защитените елементи на този клас.
Функциите-елементи на операциите се извикват само в случай, че левият операнд на бинарна операция или единственият операнд на унарна операция е обект на същия клас, елемент на който е функцията.
Друга причина за използване на предефиниращa функция, която не е елемент на класа, е когато искаме предефинираната операция да е комутативна. Обектът на класа трябва да се намира отляво на знака за операцията ако операцията е предефинирана като функция-елемент.

Затова функцията се предефинира като приятелска, за да може аргумент, който е от типа на класа, да се намира вдясно от знака на операцията. Ако операция се предефинира с обикновена функция, тогава достъпът до скритите и защитените данни-елементи на класа ще бъде неефективен.
Унарна операция може да се предефинира с помощта на нестатична функция-елемент без аргументи или с функция, която не е елемент на клас, с един аргумент. Този аргумент трябва да бъде или обект на клас, или псевдоним на обект на клас.
Предефиниращата функция-елемент трябва да е нестатична за да има достъп до нестатичните данни-елементи на класа.
Бинарна операция може да се предефинира с помощта на нестатична функция-елемент с един аргумент или с функция, която не е елемент на клас, с два аргумента. Единият от тези аргументи трябва да бъде или обект от клас, или псевдоним на обект от клас.

Примерна програма: Програмата демонстрира предефиниране на операциите ”изход на данни в поток” и ”вход на данни от поток” за обработка на данни от определен от потребителя клас за телефонни номера PhoneNumber. В тази програма се предполага, че телефонните номера се въвеждат правилно.
Функцията-операция ”вход на данни от поток” (operator>>) получава като аргументи псевдоним input на типа istream и псевдоним с име num на определения от потребителя тип PhoneNumber. Функцията връща псевдоним от тип istream.
Функцията-операция (operator>>) се използва за въвеждане на телефонни номера във вида
(800) 555-1212
в обекти от класа PhoneNumber. Когато компилаторът срещне израз
cin >> phone
в main, той генерира извикване на функцията
operator>>(cin, phone);

```
#include <iostream.h>
class PhoneNumber
 { friend ostream &operator<< (ostream &, const PhoneNumber &);
   friend istream &operator>> (istream &, PhoneNumber &);
   private:
     char areacode[4];
     char exchange[4];
     char line[5];
 } ;
ostream &operator<< (ostream &output, const PhoneNumber &num)
 { output << “(“ << num.areacode << “) “ << num.exchange << ‘-‘
          << num.line;
   return output;//осигуряваме слепване на извиквания
 }
istream &operator>> (istream &input, PhoneNumber &num)
 { input.ignore ();
   input.getline (num.areacode, 4);
   input.ignore (2);
```

```
  input.getline (num.exchange, 4);
  input.ignore ();
  input.getline (num.line, 5);
  return input;//осигуряваме слепване на извиквания
 }
void main ()
 { PhoneNumber phone;
  cout << ”Въведете номера на телефони във вид (123) 456-7890:
\n”
  cin >> phone;
  //компилаторът заменя този оператор с operator>> (cin, phone);
  PhoneNumber phone1, phone2;
  cin >> phone1 >> phone2;//слепване на извиквания
  cout << phone << endl;
  //вторият оператор << е стандартно предефинирания
 }
```

Предефинираните операции трябва да имат ляв операнд от тип ostream (istream), например cout (cin), така че предефиниращите функции не могат да бъдат функции-елементи на класа. Тези функции се нуждаят от пряк достъп до скритите данни-елементи на класа и затова се обявени като приятелски функции на този клас.
Функцията ignore е функция-елемент на класа istream, който е описан в iostream.h. Тя игнорира указания брой символи от входния поток, по подразбиране игнорира един символ.
Функцията getline е функция-елемент на класа istream. Тя прочита указаният брой символи и ги записва в низ, като автоматично добавя нулев байт.

24. Пример за клас масив

Ще създадем клас масив. Обектите на този клас ще съдържат самият масив, както и броя на елементите в него. Позволява се присвояване, сравняване, въвеждане, извеждане на масиви.

Прави се проверка дали се излиза от границите на масива.
Поддържа се статична данна-елемент, която указва броя на
създадените масиви.

```
#include <iostream.h>
#include <assert.h>
class Array
 { friend ostream &operator<< (ostream &output, const Array &);
   friend istream &operator>> (istream &input, Array &);
   public:
     Array (int = 10);
     Array (const Array &);
     ~Array ();
     int getSize () const;
     const Array &operator= (const Array &);
     int operator== (const Array &) const;
     int operator!= (const Array &) const;
     int &operator[] (int);
     static int getArrayCount ();
```

```
  private:
    int *ptr;
    int size;
    static int arrayCount;
} ;
int Array::arrayCount = 0;
Array::Array (int arraySize)
 { arrayCount++;
   size = arraySize;
   ptr = new int[size];
   assert (ptr != NULL);
   for (int i = 0; i < size; i++) ptr[i] = 0;
 }
Array::Array (const Array &init)
 { arrayCount++;
  size = init.size;
  ptr = new int[size];
  assert (ptr!=NULL);
```

```
  for (int i = 0; i < size; i++) ptr[i] = init.ptr[i];
 }
Array::~Array ()
 { arrayCount--;
   delete []ptr;
 }
int Array::getSize () const
 { return size; }
const Array &Array::operator= (const Array &right)
 { if (&right != this)
    { delete []ptr;
      size = right.size;
      ptr = new int[size];
      assert (ptr != NULL);
      for (int i = 0; i < size; i++)
       { ptr[i] = right.ptr[i]; }
    }
   return *this;
```

```
 }
int Array::operator== (const Array &right) const
 { if (size != right.size) return 0;
   for (int i = 0; i < size; i++)
    if (ptr[i] != right.ptr[i]) return 0;
   return 1;
 }
int Array::operator!= (const Array &right) const
 { return !(*this==right); }
int Array::getArrayCount ()
 { return arrayCount; }
int &Array::operator[] (int ind)
 { assert (ind>=0 && ind < size); return ptr[ind]; }
istream &operator>> (istream &input, Array &a)
 { for (int i = 0; i < a.size; i++)
    input >> a.ptr[i];
   return input;
 }
```

```
ostream &operator<< (ostream &output, const Array &a)
 { int i;
   for (i = 0; i < a.size; i++)
    { output << a.ptr[i] << ‘ ‘;
      if ( (i+1) % 10 == 0) output << endl;
    }
   if (i % 10 != 0) output << endl;
   return output;
 }
void main ()
 { cout << Array::getArrayCount () << endl;
   Array integers1 (7), integers2;
   cout << integers1.getSize () << endl;
   cout << integers1 << endl;
   cin >> integers1 >> integers2;
   cout << integers1 << integers2;
   if ( integers1 != integers2)
    cout << “Те не са равни!\n” << endl;
```

```
  Array integers3 (integers1);
  integers1 = integers2;
  integers1[5] = 1000;
  integers1[15] = 1000;
}
```

Първият конструктор Array (int = 10) ще се извиква когато компилаторът срещне изразът
Array integers (7);
Еквивалентен запис е
Array integers = 7;
Изразът
Array integers;
в този случай е еквивалентен на
Array integers (10); ⇔ Array integers = 10;
Конструкторът Array (const Array &) се нарича **конструктор за копие.** Той се извиква всеки път, когато възникне необходимост да се копира обект – например при връщане по стойност на обект от функция, при предаване по стойност на обект към функция.

Този конструктор също така се извиква при инициализиране на обект, който да е копие, например:
Array integers3 (integers1);
Еквивалентен запис е
Array integers3 = integers1;
Ще отбележим, че конструкторът за копие трябва да използва извикване по псевдоним, а не извикване по стойност. Това е така, защото при извикване по стойност активирането на конструктора ще доведе до безкрайна рекурсия, тъй при всяко извикване трябва да се създава копие и отново ще се извиква този конструктор.
Добър стил на програмиране е конструкторът, конструкторът за копие, деструкторът и предефинираната операция за присвояване да се използват съвместно за класове, които използват динамично разпределена памет.
В предефиниращата функция за операцията ‘=’ се прави проверка за самоприсвояване. При опит за самоприсвояване, присвояването се пропуска, тъй като то фактически е извършено.

Ако нямаше такава проверка, масивът десен операнд щеше да се унищожи още преди самото присвояване. Типична грешка е отсъствието на проверка за самоприсвояване при предефиниране на операция ‘=’ за клас, който съдържа указател към динамично разпределена област от паметта.
Предефиниращата функция връща като резултат левият операнд, което дава възможност за слепване. Когато срещне израза
integers1 = integers2;
компилаторът го заменя с
integers1.operator= (integers2);
Възможно е да се забрани присвояването на един обект на друг. Това се постига като предефиниращата функция за операцията ‘=’ се опише като скрит елемент на класа. Може да се забрани копирането на обекти – чрез описване на конструктора за копие и на предефиниращата функция за ‘=’ като скрити функции-елементи.

При предефинирането на операцията [], резултатът е псевдоним на елемент от масива, което позволява този резултат да се използва като ляв операнд в присвояване. Когато срещне израза
integers1[5] = 1000;
компилаторът го заменя с
integers1.operator[](5) = 1000;

25. Пример за клас низ

В C++ няма вграден тип низ. Със средствата на C++ може да се построи клас, който да управлява създаването и обработката на низове.
Отначало ще обсъдим скритите данни, използвани за представяне на обектите String. След това ще разгледаме открития интерфейс на класа, обсъждайки всяка от услугите, които предоставя нашият клас.
По-нататък ще разгледаме програмата-драйвер в main.

Ще обсъдим желания от нас стил на програмиране, т.е. вида на изразите, които бихме искали да можем да записваме за обектите от нашия нов класа String, и съответстващия набор от предефинирани операции на класа.
След това ще обсъдим дефиницията на функции-елементи на класа String. За всяка предефинирана операция ще разгледаме кода в програмата-драйвер, който активизира предефинираната функция-операция, и ще обясним как работи предефинираната функция-операция.

```
#include <iostream.h>
#include <iomanip.h>
#include <assert.h>
#include <string.h>
class String
 { friend ostream &operator<< (ostream &, const String &);
   friend istream &operator>> (istream &, String &);
   public:
    String (const char * = “”);
```

```
    String (const String &);
    ~String ();
    const String &operator= (const String &);
    const String &operator+= (const String &);
   int operator!() const;
   int operator== (const String &) const;
   int operator!= (const String &) const;
   int operator< (const String &) const;
   int operator> (const String &) const;
   int operator<= (const String &) const;
   int operator>= (const String &) const;
   char &operator[] (int);
   String operator () (int, int); //извличане на подниз
   int getLength () const;
 private:
   char *sPtr;
   int length;
} ;
```

```
String::String (const char *s)
 { length = strlen (s);
   sPtr = new char[length+1];
   assert (sPtr!=NULL);
   strcpy (sPtr, s);
 }
String::String (const String &copy)
 { length = copy.length;
   sPtr = new char [length+1];
   assert (sPtr != NULL);
   strcpy (sPtr, copy.sPtr);
 }
String::~String ()
 { delete []sPtr; }
const String &String::operator= (const String &right)
 { if (&right != this)
    { delete []sPtr;
      length = right.length;
```

```
      sPtr = new char[length+1];
      assert (sPtr != NULL);
      strcpy(sPtr, right.sPtr);
    }
   return *this;
 }
const String &String::operator+= (const String &right)
 { char *tempPtr = sPtr;
   length += right.length;
   sPtr = new char [length+1];
   assert (sPtr != NULL);
   strcpy (sPtr, tempPtr);
   strcat (sPtr, right.sPtr);
   delete []tempPtr;
   return *this;
 }
int String::operator! () const
 { return length == 0; }
```

```
int String::operator== (const String &right) const
 { return strcmp(sPtr, right.sPtr) == 0; }
int String::operator!= (const String &right) const
 { return strcmp(sPtr, right.sPtr) != 0; }
int String::operator< (const String &right) const
 { return strcmp(sPtr, right.sPtr) < 0; }
int String::operator> (const String &right) const
 { return strcmp(sPtr, right.sPtr) > 0; }
int String::operator<= (const String &right) const
 { return strcmp(sPtr, right.sPtr) <= 0; }
int String::operator>= (const String &right) const
 { return strcmp(sPtr, right.sPtr) >= 0; }
char &String::operator[] (int subs)
 { assert (subs >= 0 && subs < length);
   return sPtr[subs];
 }
String String::operator() (int index, int subl)
 { assert (index >= 0 && index < length && subl >= 0);
```

```
   String sub;
   if (subl == 0 || (index+subl > length))
    sub.length = length – index;
   else sub.length = subl;
   //приема се, че при дължина на подниза 0 се взема низа до края
   delete []sub.sPtr;
   sub.sPtr = new char[sub.length+1];
   assert (sub.sPtr != NULL);
   strncpy (sub.sPtr, sPtr + index, sub.length);
   sub.sPtr[sub.length] = '\0';
   return sub;//връща копие на sub
  }
int String::getLength () const
  { return length; }
ostream &operator<< (ostream &output, const String &s)
{  output << s.sPtr;
   return output;
}
```

```
istream &operator>> (istream &input, String &s)
{ char temp[100];
  input >> setw (100) >> temp;
  s = temp;
  return input;
}
void main ()
{ String s1 (“happy”), s2 (“ birthday”), s3;
  cout << s1 << s2 << endl << s3 << endl;
  cout << (s1==s2) << endl << (s2 < s1) << endl;
  if (!s3) cout << “s3 е празен низ!” << endl;
  s3 = s1;
  s1 += s2;
  s1 += “ to you!”;
  cout << s1 (0, 14) << endl;
  cout << s1 (15, 0) << endl;
  String *s4ptr = new String (s1);
  *s4ptr = *s4ptr;
```

```
  delete s4ptr;
  s1[0] = 'H'; s1[6] = 'B'; s1[18] = 'Y';
}
```

Резултат:

```
happy birthday
0
1
s3 е празен низ!
happy birthday to you!
```

Happy Birthday to You! – новата стойност на s1, но не отпечатана!

Компилаторът не знае как да преобразува вграден тип в тип, дефиниран от потребителя. Програмистът явно трябва да укаже как се извършват тези преобразувания – това може да стане с **конструктор за преобразуване**. Това е конструктор с един аргумент, който се използва за преобразуване на аргумента в обект от класа на конструктора. Компилаторът може да извиква такъв конструктор неявно. Обикновено предефинираната операция ‘=’ се използва за присвояване на един обект друг обект от същия клас. С помощта на конструктора за преобразуване, тя също може да се използва за присвояване на обекти от различни класове или за присвояване на един обект стойност или променлива от вграден тип.
Всеки конструктор с единствен аргумент може да се разглежда като конструктор за преобразуване.

Важно е да се отбележи, че при неявно преобразуване на типовете, дефинирано от потребителя, в C++ може да се извика само един конструктор, т.е. съгласуването на типовете в един израз не може да се получава след повече от едно преобразуване, дефинирано от потребителя. Възможно е преди да се изпълни потребителско преобразуване, компилаторът да извърши преобразуване между вградени типове и класове.
В примера конструкторът преобразува тип char* в обект от клас String и това (заедно с предефинирата операция '=') позволява на обект от клас String да се присвои масив от символи.
Конструкторът за преобразуване може да бъде извикан по следните начини:
String s1 ("Поздрав!");
String s1 = "Поздрав!";
String s1; s1 = "Поздрав!";
Конструкторът за преобразуване позволява към обект от клас String да се приложи предефинираната операция '+=' чрез низ от символи от тип char*.

Ще отбележим, че при създаване на временен обект от клас String се извиква конструктора за преобразуване, а при неговото унищожаване – деструктора. Аналогични разходи създава конструкторът за копие при предаване на обекти като параметри по стойност към функции или при връщане по стойност на обект като резултат от функция. Това води до неефективност – в горната програма, например, предефиниращата функция за операция '+=' може да приема направо аргумент от тип char*. От друга страна неявните преобразувания водят до по-малък обем на програмата и по-малко грешки в нея.

Предефинирането на операцията () е мощно средство, тъй като функциите могат да имат списъци от параметри с произволна дължина и сложност. Да отбележим, че предефиниращата функция на тази операция връща копие на локален обект, а не псевдоним, тъй като локалният обект се унищожава след напускане на функцията. Това копие ще бъде унищожено веднага след използването на резултата (подниза) в съответен израз.

Манипулаторът setw е описан в iomanip.h. Той гарантира, че няма да бъдат прочетени повече от 99 символа (добавя се нулев байт).
Операция за преобразуване може да се използва за преобразуване на обект от един клас в обект от друг клас или в обект от вграден тип. Такава операция за преобразуване трябва да бъде нестатична функция-елемент. Тя не може да бъде приятелска функция. Например следният прототип
A::operator char* () const;
декларира предефиниращa функция-операция за преобразуване, която създава временен обект от тип char* чрез преобразуване на обект от клас A. В предефиниращата функция на операция за преобразуване не се задава типа на връщания резултат, тъй като това е типът, към който се преобразува обекта.
Например, ако s е обект от клас, то когато компилаторът срещне израза (char *) s, той генерира s.operator char *();
Прототипите
A::operator int () const;
A::operator otherclass () const;

декларират предефиниращи функции-операции за преобразуване на обект от клас A в цял тип и в обект от друг потребителски тип otherclass.
Ако е необходимо, компилаторът автоматично извиква операциите и конструкторите за преобразуване при създаване на временни обекти, които да се използват в изрази, при предаване на параметри на функция или при връщане на резултат от функция. Например, ако операцията за преобразуване на обект от класа String към обект от тип char* беше определена, това позволява да не се предефинира операцията за извеждане в поток за този клас.

26. Пример за клас дата

Ще дефинираме клас Date, който използва предефинирана операция ++ за увеличаване с 1 в префиксна и постфиксна форма.

Предефиниране на ++ и - -

Всички операции за инкремент и декремент в префиксна и постфиксна форма могат да бъдат предефинирани. Ще видим, как компилаторът различава префиксните или постфиксните варианти на операциите за инкремент или декремент.
За да се предефинира операцията за инкремент за получаване на возможност за използване и на префиксна, и на постфиксна форма, всяка от тези две предефинирани функции-операции трябва да има различна сигнатура, та компилаторът да има възможност да определи каква версия на ++ се има предвид във всеки конкретен случай. Префиксният вариант се предефинира точно така, както и всяка друга префиксна унарна операция.
Да предположим например, че ние искаме да добавим 1 към деня в обекта dl на класа Date. Когато компилаторът срещне израз с префиксен инкремент
++d1

компилаторът ще генерира извикване на функция-елемент
dl.operator++ ()
прототипът на която трябва да има вида
Date operator++();
Ако префиксната форма за инкремент се реализира като функция, която не е елемент, то когато компилаторът срещне израза
++d1
той генерира извикването на функцията
operator++(dl)
прототипът на която трябва да бъде обявен в класа Date като приятелски:
friend Date operator++(Date &);
Предефинирането на постфиксната форма на операцията за инкремент представлява известна трудност, тъй като компилаторът трябва да е способен да различи сигнатурата на предефинираните функции-операции за инкремент в префиксна и постфиксна форма.

По съглашение, прието в C++, когато компилаторът срещне израз с постфиксна форма за инкремент той генерира извикване на функцията
dl.operator++(0)
прототип на която е
Date operator++(int)
Нулата (0) в генерираното извикване на функцията е чисто формална стойност, въведена за да направи списъка с аргументи на функцията operator++, използвана за постфиксната форма за инкремент, различен от списъка с аргументи на функцията operator++, използвани за префиксната форма на инкремента.
Ако постфиксната форма на операцията за инкремент се реализира като функция, която не е елемент, то когато компилаторът срещне израз
d1++
той генерира извикване на функцията
operator++(dl,0)

прототипът на която трябва да има вида
friend Date operator++(Date &, int);
Отново формалният аргумент 0 се използва от компилатора само за това, списъкът с аргументите на функцията operator++, която се използва за постфиксната форма за инкремент, да се различава от списъка с аргументите на функцията operator++, използван за префиксната форма за инкремент.
Всичко разгледано в този раздел по отношение на предефинирането на операциите за инкремент в префиксна и постфиксна форма, е приложимо и към предефинирането на операции за декремент.

```
#include <iostream.h>
class Date
{ friend ostream &operator<< (ostream &, const Date &);
  public:
    Date (int = 1, int = 1, int = 1900);
    void setDate (int, int, int);
    Date operator++ ();
    Date operator++ (int);
    const Date &operator+= (int);
    int leapYear (int) const;
    int endofMonth (int) const;
  private:
    int day; int month; int year;
    static const int days[]; void helpIncrement ();
 } ;
const int Date::days[] = { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30,
31} ;
Date::Date (int d, int m, int y)
```

```
 { setDate (d, m, y); }
void Date::setDate (int d, int m, int y)
 { month = (m>=1 && m <=12) ? m : 1;
   year = (y >= 1900 && y <=2100) ? y : 1900;
   if (month == 2 && leapYear(year))
    day = (d >= 1 && d <= 29) ? d : 1;
   else day = (d >= 1 && d <= days[month]) ? d : 1;
 }
Date &Date::operator++ ()
 { helpIncrement ();
   return *this; //връща псевдоним за създаване на L-величина
 }
Date Date::operator++ (int)
 { Date temp = *this;
   helpIncrement ();
   return temp;
 }
const Date &Date::operator+= (int addDays)
```

```
 { for (int i = 1; i <= addDays; i++)
    helpIncrement ();
   return *this; //позволява слепване
 }
int Date::leapYear (int year) const
 { if (year % 400 == 0 || (year % 100 != 0 && year % 4 == 0))
    return 1;
   return 0;
 }
int Date::endofMonth (int day) const
 { if (month == 2 && leapYear (year))
    return day == 29;
   return day == days[month];
 }
void Date::helpIncrement ()
 { if (endofMonth (day) && month == 12)
    { day = month = 1; year++; }
   else if (endofMonth(day))
```

```
   { day = 1; month++; }
  else day++;
 }
ostream &operator<< (ostream &output, const Date &d)
 { static char *monthName[13] = { “”, “Януари”, “Февруари”, “Март”,
     “Април”, “Май”, “Юни”, “Юли”, “Август”, “Септември”, “Октомври”,
     “Ноември”, “Декември” } ;
  output << d.day << ‘ ‘ << monthName[d.month] << “, “ << d.year;
  return output;
 }
void main ()
 { Date d1, d2 (27, 12, 1992), d3 (0, 99, 8045);
  cout << “d1 e ” << d1 << endl << “d2 e ”<< d2 << endl << “d3 e ”<<
d3 << endl;
  cout << “d2+=7 e ”  << (d2+=7) << endl;
  d3.setDate (28, 2, 1992);
  cout << “d3 e” << d3 << endl;
  cout << “++d3 e”<< ++d3 << endl;
```

```
  Date d4 (18, 3, 1969);
  cout  << “Проверка на операцията префиксен инкремент:” << endl
<< “d4 e ” << d4 << endl;
  cout << “++d4 e ”<< ++d4 << endl ;
  cout << “d4 e ” << d4 << endl;
  cout  << “Проверка на операцията постфиксен инкремент:” <<
endl << “d4 e ” << d4 << endl;
  cout << “d4++ e ”<< d4++ << endl ;
  cout << “d4 e ” << d4 << endl;
 }
```

За да се предефинира операцията ++ с възможност да се използва в префиксна или постфиксна форма, трябва двете предефинирани операции да имат различни сигнатури за да бъдат различавани от компилатора.

Префиксният вариант се дефинира както всяка друга префиксна унарна операция.

Например в горната програма, ако d1 е обект от клас Date и компилаторът срещне израза ++d1, той ще генерира на негово място израза d1.operator++ () прототип на която трябва да има вида Date operator++();. Префиксната форма на операцията ++ може да се реализира и с приятелска функция на класа, която се описва в дефиницията на класа така:
friend Date operator++ (Date &);
Тогава компилаторът заменя израза ++d1 с operator++ (d1).
При постфиксния вариант, уникалната сигнатура на предефинираща функция се постига с помощта на фиктивен аргумент от тип int.
Например в горната програма, ако d1 е обект от клас Date и компилаторът срещне израза d1++, той ще генерира на негово място израза d1.operator++ (0) . Прототипът на функцията е Date operator++ (int); .
Постфиксната форма на операцията ++ може да се реализира и с приятелска функция на класа, която се описва в дефиницията на класа така:

friend Date operator++ (Date &, int); .
Тогава компилаторът заменя израза d1++ с operator++ (d1, 0) .

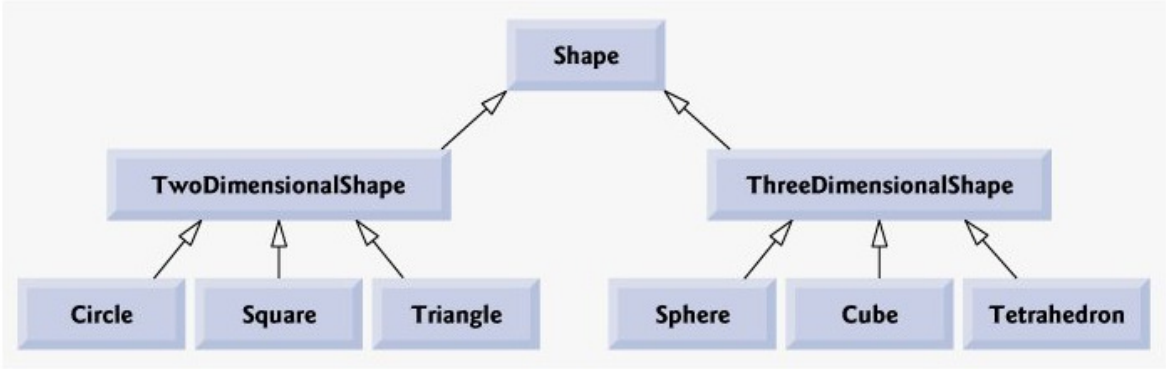
27. Наследяване. Базови и производни класове

Наследяването е начин за повторно използване на програмно осигуряване, при което новите класове се създават от вече съществуващи класове чрез заимстване на техните атрибути и функции и обогатяване на новите класове с тези възможности. Повторното използване на код икономисва време при разработката на програмите. Наследяването способства за повторното използване на проверен и тестван софтуер и по този начин намалява проблемите, възникващи след като системата започне да функционира.
Програмистът може да укаже, че създаден нов клас наследява данните-елементи и функциите-елементи на по-рано дефиниран клас.

Наследяващият клас се нарича **производен клас**, а наследеният – **базов клас**. Всеки производен клас може да се използва като базов при създаването на нови производни класове.
При **просто наследяване** производният клас наследява само един базов клас, а при **множествено наследяване** производният клас наследява два или повече базови класа.
В общия случай, производният клас добавя свои собствени данни-елементи и функции-елементи, така че обикновено той е по-голям от своя базов клас. В този смисъл, производният клас е по-специфичен по своето предназначение в сравнение със своя базов клас.
Основното предимство на наследяването е във възможността в производния клас да се определят добавки, замени или усъвършенствания на чертите, унаследени от базовия клас.
При наследяване могат да се използват всички съществуващи библиотеки от класове. В този смисъл, значителна част от програмното осигуряване може да се конструира с използването на стандартизирани компоненти.

Често обектите на един клас също са обекти и на друг клас. Правоъгълникът е също и четириъгълник (както и квадрат, и успоредник, и трапец). По такъв начин може да се каже, че класът Rectangle (правоъгълник) е наследник на класа Quadrilaterial (четириъгълник). В този контекст класът Quadrilaterial е базов клас, а класът Rectangle — производен клас. Правоъгълникът е специален тип четириъгълник, но не може да се твърди, че четириъгълникът е правоъгълник.
Наследяването формира дървовидни йерархични структури. Базовият клас се намира в йерархични отношения със своите производни класове. Класът, разбира се, може да съществува сам за себе си, но когато класът се използва в механизма на наследяването, този клас става или базов клас, който снабдява с атрибути и функции други класове, или производен клас, който наследява тези атрибути и функции.

Друга реална йерархия на наследяване е йерархията Shape.



Производният клас няма достъп до скритите елементи на своя базов клас. Разрешаването на такъв достъп би нарушило инкапсулацията на базовия клас. Производният клас, обаче, има достъп до откритите и защитените елементи на своя базов клас. Производният клас би могъл да има достъп до скритите елементи на своя базов клас, например чрез функции за достъп, предвидени в откритията интерфейс на базовия клас.

Както знаем, откритите елементи на базовия клас са достъпни за всички функции в програмата, скритите елементи на базовия клас са достъпни само за функциите-елементи и за приятелските функции на този клас. **Защитеното ниво на достъп** (**protected**) на базовия клас е междинно ниво между открития и скрития достъп. Защитените елементи на базовия клас са достъпни за елементите и приятелите на базовия клас и за елементите и приятелите на производния клас. В производния клас този достъп се осъществява само с имената на тези елементи.
Базовият клас може да се наследява от производния като public (открит), protected (защитен) или private (скрит):

- при наследяване от тип public, откритите елементи на базовия клас стават открити елементи на производния клас и защитените елементи на базовия клас стават защитени елементи на производния клас;
- при наследяване от тип protected, откритите и защитените елементи на базовия клас стават защитени елементи на производния клас;

- при наследяване от тип private, откритите и защитените елементи на базовия клас стават скрити елементи на производния клас.

И при трите вида наследявания, производният клас няма достъп до скритите елементи на базовия клас.

Базовият клас може да бъде **пряк** или **косвен** базов клас на производния клас. Прекият базов клас явно се указва в дефиницията на производния клас, за разлика от косвения базов клас, който се наследява през две или повече нива в йерархията на класовете.

При открито наследяване (от тип public), обектите на производния клас могат да се разглеждат и като обекти на базовия клас. Това не е вярно при скрито или защитено наследяване.

Обектите на базовия клас не могат автоматично да се разглеждат като обекти на производния клас. Програмистът може явно да преобразува типа на указателя към базовия клас в тип на указател към производния клас.

Типична грешка е този указател да се използва за достъп до несъществуващи елементи на производния клас.
Примерна програма: нашият пример показва дефиниция на класа Point (точка) и дефиниция на функции-елементи на класа Point, дефиниция на клас Circle (кръг) и дефиниция на функции-елементи на класа Circle, програма-драйвер, в която ние демонстрираме присвояване на указатели от производния клас на указатели от базовия клас и привеждане на типовете на указателите на базовия клас към указателите на производния клас.
Откритият интерфейс на класа Point съдържа функциите-елементи setPoint, getX и getY. Данните-елементи x и y на класа Point са указани като protected — защитени. Това забранява на клиентите на обектите на класа Point пряк достъп до данните, но позволява на класовете, породени от класа Point, непосредствен достъп до унаследените данни-елементи.

Ако тези данни бяха дефинирани като private, то за достъп до тях даже в производния клас трябваше да се използват открити функции-елементи на класа Point.

```
#include <iostream.h>
#include <iomanip.h>
class Point
 { friend ostream &operator<< (ostream &, const Point &);
   public:
     Point (float = 0.0, float = 0.0);
     void setPoint (float, float);
     float getX () const { return x; }
     float getY () const { return y; }
   protected:
     float x, y;
 } ;
Point::Point (float a, float b)
 { setPoint (a, b); }
void Point::setPoint (float a, float b)
```

```
 {  x = a; y = b; }
ostream &operator<< (ostream &output, const Point &p)
 {  output << ‘[‘ << p.x << “, “ << p.y << ‘]’;
    return output;
 }
class Circle:public Point
//класът Circle открито наследява класа Point; тук ‘:’ е указание за
//наследяване, ключовата дума public указва типа на
наследяването
 {
   friend ostream &operator<< (ostream &, const Circle &);
   public:
     Circle (float r = 0.0, float x = 0.0, float y = 0.0);
     void setRadius (float); // задаване на радиуса
     float getRadius () const; // получаване на радиуса
     float area () const; // изчисляване на лицето
   protected:
     float radius;
```

```
 } ;
Circle::Circle (float r, float a, float b)
: Point (a, b) // извикване на конструктора на базовия клас
 { setRadius ( r ); }
void Circle::setRadius (float r)
 { radius = (r >= 0 ? r : 0) ; }
float Circle::getRadius () const
 { return radius; }
float Circle::area () const
 { return 3.14159*radius*radius; }
ostream &operator<< (ostream &output, const Circle &c)
 { output << "Центърът = [" << c.x << ", " << c.y << "]; Радиусът = "
          << setiosflags (ios::showpoint) << setprecision (2) << c.radius;
   return output;
 }
void main ()
{ Point *pointPtr = 0, p (3.5, 5.3);
  Circle *circlePtr = 0, c (2.7, 1.2, 8.9);
```

```
  cout << “Точка p: “ << p << endl << “Окръжност c: “ << c << endl;
  pointPtr = &c;
  cout << “Окръжност c (c *pointPtr) : “<< *pointPtr << endl;
  circlePtr = (Circle *) pointPtr;
  cout << “Окръжност c (c *circlePtr) : “<< *circlePtr << endl;
  cout << “Лицето c (c *circlePtr) : “ << circlePtr->area () << endl;
  pointPtr = &p;
  circlePtr = (Circle *) pointPtr;
  cout << “Точка p (c *circlePtr) : “<< endl << “Лицето на обекта
circlePtr сочи към: “ << circlePtr->area () << endl;
}
```

Привеждане на типовете на указателите на базовия клас към типовете на указателите на производния клас

Резултат:
Точка p: [3.5, 5.3]
Окръжност c: Център = [1.2, 8.9]; Радиус =2.70
Окръжност c (c *pointPtr): [1.20, 8.90]
Окръжност c (c *circlePtr):
Център = [1.20, 8.90]; Радиус =2.70
Лицето c (c circlePtr): 22.90
Точка p (c *circlePtr):
Център = [3.50, 5.30]; Радиус = 4.02e-38
Лицето на обекта circlePtr сочи към: 0.00

Ще отбележим, че конструкторът на Circle извиква конструктора на Point за да зададе начални стойности на частта от обекта на класа Circle, която се отнася до базовия клас Point. Това става чрез списък от инициализатори на елементи

```
Circle::Circle (float r, float a, float b)
: Point (a , b)
```

Ако конструкторът на Circle не активира конструктора на Point явно, то конструкторът на Point се активира със стойности по премълчаване. Ако класът Point няма конструктор с елементи по премълчаване, данните-елементи на класа Point ще имат неопределени стойности.
Предефиниращата функция на операцията за извеждане в поток на класа Circle има непосредствен достъп до защитените данни-елементи x и y на класа Point, тъй като тази функция е приятелска на производния клас Circle.
Указател към производния клас се присвоява винаги правилно на указател към базовия клас, тъй като обектите на производния клас могат да се разглеждат като обекти на базовия клас.
Компилаторът извършва такова преобразуване неявно. Указателят към базовия клас вижда само тази част от производния клас, която се отнася към базовия клас. Такова присвояване може да стане и чрез псевдоним, например: Point &pRef = c – дефиниране на псевдоним на обект от базовия клас Point, който става псевдоним на обект от производния клас Circle.

Присвояването на указател към базовия клас на указател към производния клас е опасно и в този случай компилаторът не извършва неявно преобразуване – затова трябва да се използва явно такова. Извеждането на обект от клас Point като обект от клас Circle води до получаване на неопределена стойност за данната-елемент radius. В общия случай, достъпът до несъществуващи данни-елементи не е опасен, но извикването на несъществуваща функция-елемент може да доведе до необратими грешки в програмата.
Възможно е с помощта на предефинирана операция за присвояване или конструктор за преобразуване на обект от производен клас да се присвои обект от съответен базов клас. Такова присвояване в общия случай би оставило неопределени собствените елементи на производния клас.

28. Предефиниране на функции-елементи от базовия клас в производния клас

Производният клас може да предефинира функции-елементи на базовия клас. Ако в производния клас се опише функция-елемент със същото име както функция-елемент на базовия клас, то версията на тази функция в производния клас предефинира версията от базовия клас. За да се направи достъпна версията на функцията от базовия клас за производния клас, трябва да се използва бинарната операция за разрешаване на област на действие ‘::’.
Примерна програма: ще разгледаме опростен клас Employee (служещи). Той съхранява имената firstName и фамилиите lastName на служещите. Тази информация е обща за всички служещи, включително намиращите се в класовете, породени от класа Employee.

На основа на класа Employee са породени класовете HourlyWorker (работници с почасова заплата), Pieceworker (работници на парче), Boss (управляващи) и CommissionWorker (работници на комисионна). За простота ще изучим само класа Employee и производния клас HourlyWorker.

```
#include <string.h>
#include <iostream.h>
#include <iomanip.h>
#include <assert.h>
class Employee {
  public:
    Employee (const char *, const char *);
    void print () const;
    ~Employee ();
  private:
    char *firstName, *lastName;
} ;
Employee::Employee (const char *first, const char *last)
```

```
 { firstName = new char [ strlen (first) + 1];
   assert (firstName != NULL);
   strcpy (firstName, first);
   lastName = new char [ strlen (last) + 1];
   assert (lastName != NULL);
   strcpy (lastName, last);
 }
void Employee::print () const
 { cout << firstName << ' ' << lastName; }
Employee::~Employee ()
 { delete [] firstName; delete [] lastName; }
class HourlyWorker : public Employee
 { public:
     HourlyWorker (const char *, const char *, float, float);
     float getPay () const;
     void print () const;
  private:
     float wage, hours;
```

```
 } ;
HourlyWorker::HourlyWorker (const char *first, const char *last,
float initHours, float initWage) : Employee (first, last)
 { hours = initHours; wage = initWage; }
float HourlyWorker::getPay () const
 { return wage*hours; }
void HourlyWorker::print () const
 {cout << “HourlyWorker::print () се изпълнява\n\n”;
 Employee::print (); // извикване на функцията print от базовия клас
   cout << “ – работник с почасово заплащане $” << setiosflags
(ios::fixed | ios::showpoint)
          << setprecision (2) << getPay () << endl;
 }
void main ()
 { HourlyWorker h ( “Иван”, “Петров”, 40, 7.50);
   h.print ();
 }
```

Резултат:

HourlyWorker h(“Иван”, “Петров”, 40, 7.50);
h.print ();

HourlyWorker::print()
Иван Петров – работник с почасово заплащане заплата $300.00

Функцията-елемент print на класа HourlyWorker е пример за предефинирана функция-елемент на базов клас в производен клас.
Функции-елементи от базовия клас често се предефинират в производния клас за изпълняване на някои по-специфични действия. Тези предефинирани функции понякога извикват версията на функцията от базовия клас, за да изпълнят част от новата задача. В този пример функцията print от производния клас извиква функцията print от базовия клас за да изведе името на служещия (функцията print от базовия клас е единствената функция, имаща достъп до скритите данни на базовия клас). Функция print от производния клас извежда също заплатата на служещите. Функцията print от базовия клас се извиква по следния начин:
Employee::print();
Тъй като функциите от базовия клас и производния клас имат еднакви имена и сигнатури, функциите от базовия клас трябва да се предшестват от името на класа и операцията за разрешаване

на областта на действие. В противен случай ще се извика пак версията на функцията на производния клас, което ще доведе до безкрайна рекурсия (функцията print от класа HourlyWorker ще се извиква сама себе си).

Открити, защитени и скрити базови класове

При пораждане на клас от базов клас, базовият клас може да се наследява като public, protected или private. Защитеното и скритото наследяване се срещат рядко и всяко от тях трябва да се използва с голямо внимание. Обикновено се използва само откритото наследяване.

При пораждане на клас като public, откритите елементи на базовия клас стават открити елементи на производния клас, а защитените елементи на базовия клас стават защитени елементи на производния клас. Скритите елементи на базовия клас никога не са достъпни за производния клас.

При защитеното наследяване откритите и защитените елементи на базовия клас стават защитени елементи на производния клас.

При скритото наследяване откритите и защитените елементи на базовия клас стават скрити елементи на производния клас. При скритото и защитеното наследяване не е вярно, че обектът от производния клас е обект на базовия клас.
Таблицата обобщава достъпа на елементите на базовия клас от производния клас, основаващ се на спецификаторите за достъп до елементите в базовия клас и типа на наследяването. Първият стълб съдържа спецификатори за достъп до елементите в базовия клас. Първият ред съдържа типовете наследяване. Останалата част от таблицата указва спецификаторите за достъп до елементите на базовия клас, които са приложими в производния клас, и кратко описание как може да се осъществи достъп до елементите на базовия клас.

Спецификатор за достъп до елементите в базовия клас	Тип наследяване		
	public открито наследяване	protected защитено наследяване	private скрито наследяване
public	public в производния клас Може да бъде дос- тъпен непосредст- вено за всякакви нестатични функции-	protected в производния клас Може да бъде дос- тъпен непосредст- вено за всякакви нестатични функции-	private в производния клас Може да бъде дос- тъпен непосредст- вено за всякакви нестатични функции-

	елементи, на приятелски функции и функции, които не са елементи	елементи и на приятелски функции	елементи и на приятелски функции
protected	protected в производния клас Може да бъде дос- тъпен непосредст- вено за всякакви нестатични	protected в производния клас Може да бъде дос- тъпен непосредст- вено за всякакви нестатични	private в производния клас Може да бъде дос- тъпен непосредст- вено за всякакви нестатични

	функции-елементи и на приятелски функции	функции-елементи и на приятелски функции	функции-елементи и на приятелски функции
private	невидим в производния клас Може да бъде дос-тъпен за нестатични функции-елементи и на приятелски с открити или	невидим в производния клас Може да бъде дос-тъпен за нестатични функции-елементи и на приятелски с открити или	невидим в производния клас Може да бъде дос-тъпен за нестатични функции-елементи и на приятелски с открити или

	защитени функции-елементи на базовия клас	защитени функции-елементи на базовия клас	защитени функции-елементи на базовия клас

Преки и косвени базови класове

Базовият клас може да е пряк или косвен базов клас на производния клас. Прекият базов клас явно се записва в заглавието при обявяването на производния клас. Косвеният базов клас не се записва в заглавието на производния клас. Той се наследява през две или повече нива на йерархия на класовете.

29. Използване на конструктори и деструктори в производни класове

Тъй като производния клас наследява елементите на базовия клас, то при създаването на обект на производния клас трябва да бъде извикан конструкторът на базовия клас за задаване на началните стойности на елементите на базовия клас, съдържащи се в обекта от производния клас. В конструктора на производния клас при явно извикване на конструктора на базовия клас може да бъде предвиден списък от инициализатори на елементите; в противен случай конструкторът на производния клас неявно ще извиква конструктора на базовия клас по премълчаване. Конструкторите и операциите за присвояване не се наследяват от производните класове. Обаче, конструкторите и операциите за присвояване на производния клас могат да извикват конструкторите и операциите за присвояване на базовия клас.

Конструкторът на производния клас винаги най-напред извиква конструктора на своя базов клас за задаване на началните стойности на тези елементи от производния клас, които са идентични на елементите от базовия клас. Ако конструкторът на производния клас отсъства, то конструкторът по премълчаване на производния клас извиква конструкторът на базовия клас.
Деструкторите се извикват в ред, обратен на извикванията на конструкторите, така че деструкторът на производния клас се извиква преди съответния деструктор на базовия клас.
При създаване на обект от производен клас, първо се изпълнява конструкторът на базовия клас, след това конструкторите на обектите-елементи на производните класове и най-накрая се изпълнява самият конструктор на производния клас.
Деструкторите се извикват в ред, обратен на този, в който са извикани съответните конструктори.
Редът, в който се извикват конструкторите на обектите-елементи, е редът, в който те са описани в дефиницията на класа.

На този ред не влияе редът, в който са изброени инициализаторите на елементите.
При наследяването конструкторите на базовите класове се извикват в последователността, в която е указано наследяването в дефиницията на производния клас. На този ред не влияе последователността, в която са посочени конструкторите на базовите класове в описанието на конструктора на производния клас.
Примерна програма: демонстрира се последователността, в която се извикват конструкторите и деструкторите на производния клас. Показва се прост клас Point, съдържащ конструктор, деструктор и защитени данни-елементи x и y. Конструкторът и деструкторът печатат обект на класа Point, за който те са активирани.
След това се показва прост клас Circle, наследяващ Point с открито наследяване, съдържащ конструктор, деструктор и скрита данна-елемент radius. Конструкторът и деструкторът печатат обект на класа Circle, за който те са активирани.

Конструкторът Circle активира също конструкторът на класа Point, използвайки списък от инициализатори на елементи, и предава стойностите на a и b за задаване на начални стойности на данните-елементи на базовия клас.

```
#include <iostream.h>
class Point {
 public:
    Point (float = 0.0, float = 0.0);
    ~Point ();
 protected:
    float x, y;
 } ;
Point::Point (float a, float b)
 {  x = a; y = b;
    cout << “Конструктор на Point: “ << ‘[‘ << x << “, “ << y << ‘]’ <<
endl;
 }
Point::~Point ()
```

```
 {  cout << “Деструктор на Point: “ << ‘[‘ << x << “, “ << y << ‘]’ << endl;
}
class Circle : public Point
 {  public:
      Circle (float r = 0.0, float x = 0.0, float y = 0.0);
      ~Circle ();
    private:
      float radius;
 } ;
Circle::Circle (float r, float a, float b) :
   Point (a, b) // извикване на конструктора на базовия клас
 { radius = r;
   cout << “Конструктор на Circle: радиусът е “ << radius << ‘[‘ << x <<
“, “ << y << ‘]’
           << endl;
 }
Circle::~Circle ()
```

```
 { cout << “Деструктор на Circle: радиусът е “ << radius << ‘[‘ << x <<
“, “ << y << ‘]’
          << endl;
 }
void main ()
 {
   { Point p (11., 22.); }
     cout << endl;
     Circle circle1 (4.5, 72., 29.);
     cout << endl;
     Circle circle2 (10., 5., 5.);
 }
```

Резултат от изпълнението:

```
Конструктор на Point: [11., 22.]
Деструктор на Point: [11., 22.]
Конструктор на Point: [72., 29.]
```

Конструктор на Circle: радиусът е 4.5 [72., 29.]
Конструктор на Point: [5., 5.]
Конструктор на Circle: радиусът е 10. [5., 5.]
Деструктор на Circle: радиусът е 10. [5., 5.]
Деструктор на Point: [5., 5.]
Деструктор на Circle: радиусът е 4.5 [72., 29.]
Деструктор на Point: [72., 29.]

30. Множествено наследяване

При простото наследяване, всеки производен клас се поражда само от един базов клас. Йерархията при простото наследяване може да се изобрази чрез дърво. При множественото наследяване, производният клас се поражда от два или повече базови класа. Йерархията при множественото наследяване може да се изобрази чрез ориентиран ацикличен граф. Йерархията на наследяването може да бъде с произволна дълбочина в границите на физическите ограничения за конкретната система.

Примерна програма: ще разгледаме пример на множествено наследяване. Класът Base1 съдържа една защитена данна-елемент — int value. Base1 съдържа конструктор, който задава стойността на value, и открита функция-елемент getData, която връща стойността на value.

```
#include <iostream.h>
class Base1 {
  public:
      Base1 (int x) { value = x; }
      int getData () const { return value; }
  protected:
      int value;
} ;
class Base2 {
 public:
     Base2 (char c) { letter = c; }
     char getData() const { return letter;}
 protected:
```

```
    char letter;
} ;
class Derived: public Base1, public Base2
{ friend ostream& operator<< (ostream &, const Derived &);
  public:
     Derived (int, char, float);
     float getReal () const;
  private:
     float real;
} ;
Derived::Derived ( int i, char c, float f) : Base1 (i), Base2 (c),  real (f) {}
float Derived::getReal () const  { return real; }
ostream &operator<< (ostream &output, const Derived &d)
 {  output << “Цяло: “ << d.value << endl
            << “Символ: “ << d.letter << endl
            << “Реално: “ << d.real << endl;
    return output;
 }
```

```
void main ()
 { Base1 b1(10), *base1Ptr = 0;
   Base2 b2 (‘Z’), *base2Ptr = 0;
   Derived d (7, ‘A’, 3.5);
  // печат на данни-елементи на обекти на базовия клас
   cout << " Обектът b1 съдържа цяло "
          << b1.getData () << endl
          << "Обектът b2 съдържа символ "
          << b2.getData () << endl
          << "Обектът d съдържа:" << endl << d << endl << endl;
  // печат на данни-елементи на обекти на производния клас
  // операцията за разрешаване на областта на действие
  // отстранява неопределеността на getData
   cout << "Данните-елементи на класа Derived могат да бъдат "
          << "достъпни индивидуално:" << endl
          << " Цяло: " << d. Base1:: getData () << endl
          << " Символ: " << d.Base2: : getData () << endl
          << " Реално: " << d.getReal() << endl << endl;
```

```
  cout << "Derived може да се разглежда като обект "
       << "от двата базови класа:" << endl;
 // разглеждане на Derived като обект от класа Base1
  base1Ptr = &d;
  cout << "Резултат от base1Ptr->getData() : "
       << base1Ptr->getData() << endl;
 // разглеждане на Derived като обект от класа Base2
  base2Ptr = &d;
  cout << "Резултат от base2Ptr->getData() : "
       << base2Ptr->getData() << endl;
}
```

```
Резултат:
Обектът b1 съдържа цяло 10
Обектът b2 съдържа символ Z
Обектът d съдържа:
Цяло: 7
Символ: A
```

Реално: 3.5

Данните-елементи на класа Derived могат да бъдат достъпни индивидуално:
Цяло: 7
Символ: A
Реално: 3.5
Derived може да се разглежда като обект от двата базови класа:
Резултат от base1Ptr->getData() : 7
Резултат от base2Ptr->getData() : A
Класът Base2 е аналогичен на класа Base1, с изключение на това, че негова защитена данна-елемент е char letter. Base2 има също открита функция-елемент get Data, но тази функция връща стойността на char letter.

В примера класът Derived се поражда от два класа Base1 и Base2 чрез множествено наследяване – то се указва с ':' след името на класа и следващ списък на базовите класове, разделени със запетаи.
Конструкторът на Derived извиква явно конструкторите на всеки от своите базови класове Base1 и Base2 с използване на списък от инициализатори на елементи. Както знаем, конструкторите на базовите класове се извикват в този ред, в който е определено наследяването, а не в този, в който конструкторите са описани.
Тъй като функцията operator<< е приятелска за производния клас Derived, тя има пряк достъп до защитените елементи value и letter от базовите класове Base1 и Base2.
По-нататък възниква проблем - един обект от клас Derived съдържа функции с еднакви имена. Това са наследените функции getData от базовите класове Base1 и Base2. Този проблем се решава лесно с помощта на бинарната операция за разрешаване на област на действие.

31. Виртуални функции и полиморфизъм

С помощта на виртуални функции и полиморфизъм могат да се разработват и реализират програми за обобщена обработка на обекти от всички съществуващи в йерархията класове, като обекти от базовия клас. Ако по време на разработката на такава програма трябва да се добави нов клас в йерархията, това може да стане с незначителни изменения на самата програма. Единствените части на програмата, които трябва да се модифицират, са частите, изискващи непосредствено познаване на конкретния клас, добавян в йерархията.
Един от начините за специфична обработка на обекти от различни класове е използването на оператор switch, който може по различен начин да обработва различните обекти в зависимост от техните типове.

Например, в йерархията на фигурите, в която всяка фигура съхранява своя тип в някакво поле — данна-елемент, структурата за избор switch е способна да определи по стойността на това поле, коя от функциите print трябва да се извика за работа с обект от даден тип. Във връзка с това, обаче, възникват различни проблеми – например липса на проверка на типа на обекта, липса на проверка за всички възможни случаи в оператора switch. Освен това, добавянето или изключването на клас води до промяна на съответния оператор switch.
Виртуалните функции и полиморфното програмиране могат да премахнат нуждата от използване на логиката на оператора switch. Програмистът може да използва механизма на виртуалните функции за автоматично изпълнение на логиката на оператора switch, избягвайки горепосочените проблеми и неудобства.

Използването на виртуални функции и полиморфизъм води до създаване на програми с по-прост вид (линейна последователност от оператори), които имат по-малко логически разклонения и по този начин се улеснява проверката и съпровождането на тези програми.
Да предположим, че редицата класове за фигури, такива като Circle, Triangle, Square и т.н. са производни от базовия клас Shape. Всеки клас може да има своя собствена функция Draw за чертане на фигурата. В този случай е удобно при чертането на всяка фигура да можем да извикваме функцията Draw за базовия клас Shape и тогава програмата динамично, т.е. по време на своето изпълнение, да определи коя от функциите Draw на съответния производен клас да се изпълни. За да се създаде такава възможност, трябва функцията Draw да се декларира като виртуална в базовия клас и след това да се предефинира във всеки от производните класове, така че да чертае съответната фигура.

Една функция се декларира като виртуална с помощта на ключовата дума virtual, записана преди прототипа на функцията в базовия клас. Например в базовия клас Shape можем да запишем:
virtual void Draw () const;
Този прототип декларира, че функцията Draw е константна функция, която няма аргументи, не връща резултат и е виртуална функция.
Ако една функция веднъж е декларирана като виртуална, то тя остава виртуална на всяко по-ниско ниво в йерархията. Ако производният клас няма собствена реализация на виртуалната функция, то се използва реализацията, описана в базовия клас. Ако функцията Draw от базовия клас е декларираната като virtual и ако след това тя се извика чрез указател от базовия клас, сочещ към обект от производния клас, например
shapePtr -> Draw ();
то програмата динамично, т.е. по време на изпълнение, ще избере съответната функция от производния клас. Това се нарича **динамично свързване.**

Предефинирата виртуална функция в производния клас трябва да има същия тип на резултата и същата сигнатура, както виртуалната функция в дефиницията в базовия клас. В противен случай се дава съобщение за синтактична грешка.
Ако виртуална функция се извиква чрез обръщение по име и при това се използва операцията за достъп до елемент ‘.’, например
squareObject.Draw();
то извикването се обработва по време на компилация и това се нарича **статично свързване.**
Полиморфизмът е възможност за обектите от различни класове, свързани с помощта на наследяване, да реагират по различен начин при обръщение към една и съща функция-елемент.
Полиморфизмът се реализира с помощта на виртуални функции.
Понякога функция-елемент, която не е дефинирана като виртуална в базовия клас се предефинира в производния клас.
Ако такава функция-елемент е извикана чрез указател от базовия клас, то се използва версията на функцията в базовия клас.

Ако е извикана чрез указател от производния клас, то се използва версията на функцията от производния клас. Това не е полиморфно поведение.
Например: имаме базов клас Employee и производен клас HourlyWorker
Employee e, *ePtr = &e; // дефинираме обект и указател от базовия клас
HourlyWorker h, *hPtr = &h; // дефинираме обект и указател от производния клас
Функцията print е невиртуална, дефинирана в базовия клас и предефинирана в производния клас.
ePtr -> print (); // извиква се функцията print от базовия клас
hPtr -> print (); // извиква се функцията print от производния клас
Функцията print от базовия клас също така е елемент от производния клас, но при извикването на функцията print за обект от производния клас вариантът на функцията от базовия клас трябва да бъде извикан явно, както е показано по-долу:
hPtr->Employee::print( ) ;

Този запис определя по явен начин, че трябва да се извика функцията print от базовия клас.
Причината, че това не е полиморфно поведение – функциите не са виртуални и имат еднаква сигнатура. Ако функцията print беше дефинирана като virtual в базовия клас, то при обръщението
ePtr -> print ();
щеше да се извика функцията елемент print от производния клас.
По време на компилация не е необходимо да се знае типа на обекта, за да се генерира извикване на виртуална функция. По време на изпълнението на програмата, чрез динамично свързване, извикването на виртуална функция-елемент се преобразува в извикване на вариант на виртуалната функция от съответния клас.
За целта се използва **таблица на виртуалните методи.** Всеки клас, който съдържа виртуални функции, има своя таблица на виртуалните методи. За всяка виртуална функция-елемент от класа, таблицата има елемент, съдържащ указател към вариант на виртуалната функция, който се използва с обектите от дадения клас. Всеки обект от клас, който съдържа виртуални функции, има

указател към таблицата на виртуалните методи на този клас. При динамичното свързване, чрез този указател се отива в таблицата на виртуалните методи и там се намира нужният указател към вариант на виртуалната функция от съответния клас.

32. Абстрактни базови класове и конкретни класове

Има случаи, в които е полезно да се дефинират класове, за които програмистът няма да създава обекти. Такива класове се наричат **абстрактни класове.** Тъй като те се използват като базови класове в процеса на наследяване, обикновено се наричат абстрактни базови класове. Обектите на абстрактните базови класове не могат да бъдат реализирани, т.е. не съществуват обекти от абстрактни базови класове.
Единственото предназначение на абстрактен клас е създаване на съответен базов клас, от който други класове могат да наследяват интерфейс и реализация на производните от тях класове. Класове, чиито обекти могат да бъдат реализирани, се наричат **конкретни класове.**

Например, можем да имаме абстрактен базов клас TwoDimensionalObject и производни конкретни класове Square, Circle, Triangle и т.н.
Абстрактните базови класове са твърде общи за дефиниране на реални обекти. За това са предназначени конкретните класове – те притежават необходимата определеност и специфика за дефиниране на реални обекти.
Един клас става абстрактен чрез дефиниране на една или повече виртуални функции като **чисто виртуални.** Една виртуална функция става чисто виртуална, когато в нейната декларация тялото е дефинирано като 0, т.е. инициализаторът е равен на 0.
Например:
virtual float earnings () const = 0;
Ако един клас е производен от клас с чисто виртуална функция и ако тази чисто виртуална функция не е описана в този клас, тогава функцията остава чисто виртуална и в производния клас. Следователно такъв производен клас също е абстрактен клас.

При опит за създаване на обект от абстрактен клас се дава съобщение за синтактична грешка.
Абстрактният базов клас определя интерфейса за различни типове обекти в йерархията на класовете. Всички обработки в йерархията могат да прилагат един и същи интерфейс, използвайки полиморфизъм – дефинират се указатели от абстрактния базов клас и след това те се използват за полиморфно опериране с обектите на производните конкретни класове.
Полиморфизмът е особено ефективен при реализацията на системно програмно осигуряване. Например при операционните системи всеки тип физическо устройство може да работи по съвършено различен начин. Независимо от това, командите за четене ‘read’ и командите за писане ‘write’ могат да бъдат подобни. Например съобщението ‘write’, изпратено на обект драйвер за устройство се интерпретира по различен начин в зависимост от типа на драйвера и от това по какъв начин той управлява съответното устройство.

Въпреки това, извикването ‘write’ е почти едно и също за различните устройства – при това извикване просто се копират определен брой байтове от паметта в конкретното устройство. Обектно-ориентираните операционни системи могат да използват абстрактни базови класове за да се реализира интерфейс, пригоден за драйверите на всички устройства. След това, с помощта на наследяване, тези абстрактни класове могат да образуват производни класове. Откритият интерфейс към драйверите на устройствата се осигурява с помощта на чисто виртуални функции на абстрактните базови класове. Реализацията на тези виртуални функции се осигурява в производните класове и съответства на конкретните типове драйвери за устройства. Полиморфизмът се използва и при класове итератори – например при обхождане в свързан списък на обекти от различни нива в йерархията. Всички указатели в такъв списък могат да бъдат указатели от базовия клас.
Конструкторите не могат да бъдат виртуални функции. Типична грешка е дефинирането на конструктор като виртуална функция.

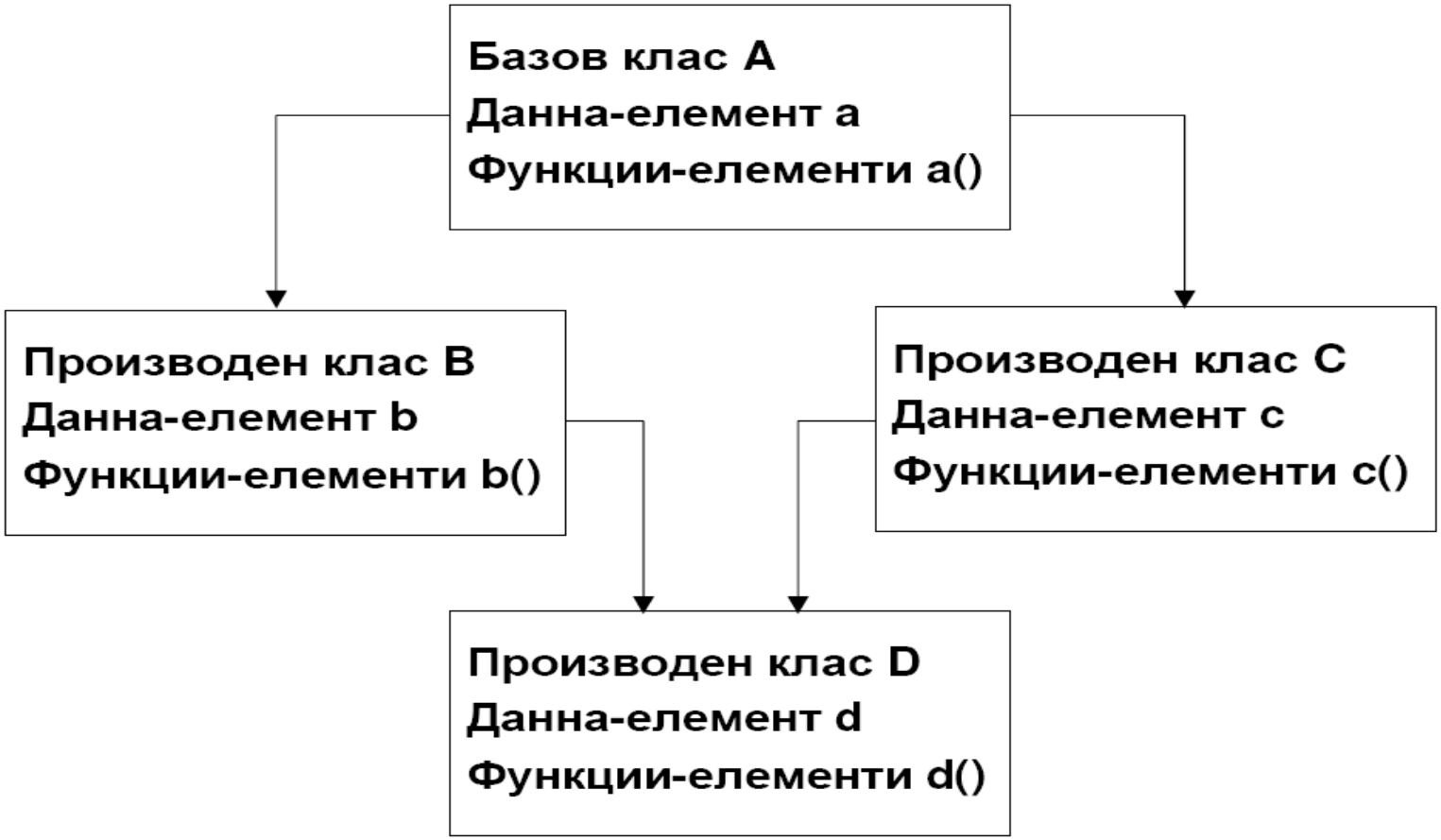
Виртуални деструктори

При използване на полиморфизма за обработка на динамично разпределени обекти от йерархията на класовете, може да се появи проблем. Ако обектът се унищожава с явно използване на операцията delete над указател от базовия клас към обекта, то се извиква деструктора на базовия клас на дадения обект. Това става независимо от типа на обекта, към който сочи указателя от базовия клас, и независимо от факта, че деструкторите на всеки клас имат различни имена.
Съществува просто решение на този проблем: обявяване на деструктора на базовия клас като виртуален. Това автоматично ще доведе до там, че всички деструктори на производните класове ще станат виртуални, даже ако те имат имена, различни от името на деструктора на базовия клас.

В този случай, ако обекта в йерархията е унищожен с явно използване на операцията delete, приложена към указателя от базовия клас към обекта от производния клас, то ще се извика деструктора от съответния клас. Да си спомним, че когато производния клас е унищожен, частта от базовия клас, съдържаща се в производния клас, също се унищожава. Деструкторът от базовия клас автоматично се изпълнява след деструктора на производния клас.
Добър стил на програмиране е когато в един клас има виртуални функции, да се предвиди създаване на виртуален деструктор, даже ако той не е необходим за този клас. Класовете, производни от даден клас, могат да съдържат деструктори, които трябва да се извикват по съответен начин.
Благодарение на виртуалните функции и полиморфизма, програмистът може да управлява широк спектър от обекти без да знае за тяхните типове. Управлението по време на изпълнението на програмата автоматично ще отчита спецификата на тези обекти.

Разрешаване на област на действие на косвено наследени елементи

Множественото наследяване може да доведе до сложни зависимости между класовете. В частност, при йерархия на класовете на много нива може да възникне ситуация на повторно косвено наследяване на един клас-прародител, например:


Базов клас A
Данна-елемент a
Функции-елементи a()
Производен клас B
Данна-елемент b
Функции-елементи b()
Производен клас C
Данна-елемент c
Функции-елементи c()
Производен клас D
Данна-елемент d
Функции-елементи d()

Тук класовете B и C са производни от един базов клас A и едновременно се използват като базови за създаване на производен клас D. По такъв начин, класът D два пъти наследява елементи от прародителския клас A, което води до появяване в обектите на класа D на двойки едноименни елементи, наследени от A или чрез B, или чрез C. За да се обърнем към такива „сдвоени” наследени елементи е необходимо да се използва операцията за разрешаване на областта на видимост за отстраняване на нееднозначността. Например, B::a и C::a са два различни елемента, косвено наследени от класа D.
Ще покажем пример на реализация на разгледаната йерархия на класовете:

```
// Файл abcd.h
#ifndef  _ABCD_H
#define  _ABCD_H
// Първи базов клас
class A{
protected:
int a;
public:
A(int);
virtual void show();
};
// Първи производен клас
class B:public A{
protected:
int b;
public:
B(int, int);
virtual void show();
```

```
};
// Втори производен клас
class C:public A{
protected:
int c;
public:
C(int, int);
virtual void show();
};
// Трети производен клас с множествено наследяване
class D: public B, public C {
int d;
public:
D(int, int, int, int, int);
void show();
};
#endif _ABCD_H
```

Общият брой на променливите от производния клас D е пет: B::a, C::a, b, c и d. Съответно за тяхната инициализация ще трябва конструктор с пет параметри.
Файл за реализация на класовете:

```
// Файл abcd.cpp
#include “abcd.h”
#include <iostream.h>
// Конструктори
A::A(int n):a(n) {}
B::B(int n, int m):A(n), b(m) {} //първият параметър задава стойност за
// конструктора на базовия клас A
C::C(int n, int m):A(n), c(m) {}
D::D(int k1, int k2, int l, int m, int n):
B(k1,l), C(k2,m), d(n) {}
// Методи
void A::show() {cout << “ a=” << a << endl;}
void B::show() {cout << “ a=” << a << “ b=” << b << endl;}
```

```
void C::show() {cout << " a=" << a << " c=" << c << endl;}
void D::show()
{cout << " B::a=" << B::a << " C::a=" << C::a
<< " b=" << b << " c=" << c << " d=" << d << endl;
}
```

Обърнете внимание на списъка за инициализация на конструктора на класа D: тук освен инициализацията на собствената данна-елемент d се извикват конструкторите на двата непосредствени родителски класове, всеки от които запълва собствена данна-елемент (b или c), а също наследената от общия базов клас една от едноименните данни-елементи (B::a или C::a съответно). Главната функция отново демонстрира работата с динамични обекти – възможност за обръщение към обектите от производните класове по указатели към базовия клас. Обаче в дадения случай има изключение: указателят от базовия клас, няколко пъти косвено наследен от производния клас, не може да се използва за обръщение към обектите от този производен клас.

Например, в нашия случай указателя от тип A* не може да се използва за обектите от класа D , което е указано в коментарите:

```
#include “abcd.h”
void main() {
A *a = new A(1);
a->show();
delete a;
a = new B(2,3);
a->show();
delete a;
a = new C(4,5);
a->show();
delete a;

// a = new D(6, 7, 8, 9, 10); // Така не може да се обръщаме
B *b = new D(11, 12, 13, 14, 15);
b->show();
delete b;
```

```
C *c = new D(16, 17, 18, 19, 20);
c->show();
delete c;
}
```

Резултат:

```
a=1
a=2 b=3
a=4 c=5
B::a=11 C::a=12 b=13 c=14 d=15
B::a=16 C::a=17 b=18 c=19 d=20
```

Виртуални класове

При множественото наследяване често е необходимо да забраним наследяването на няколко екземпляра от базовия клас. Затова всички производни класове трябва да го обявят за виртуален базов клас. Да се върнем към разглежданата йерархия. Ако логиката на нейното построяване изисква да се наследи само един екземпляр от базовия клас A в класа D и съответно да има по един екземпляр от данни-елементи и функции-елементи (в частност данната-елемент a), то непосредствените наследници на класа A (B и C) го декларират като виртуален базов клас, след което техния наследник D ще съдържа само един екземпляр от данните, наследени от A. Ще приведем новия вариант на разглежданата програма (по-точно само променената част от кода):

```
// Първи производен клас
// (базовият клас се декларира като виртуален)
class B:virtual public A{
```

```
protected:
int b;
public:
B(int, int);
virtual void show();
};
// Втори производен клас
// (базовият клас се декларира като виртуален)
class C:virtual public A{
protected:
int c;
public:
C(int, int);
virtual void show();
};
// Трети производен клас с множествено наследяване
class D: public B, public C {
int d;
```

```
public:
D(int, int, int, int);
void show();
};
```

Ще отбележим, че използването на ключовата дума virtual се изисква при деклариране само на непосредствените наследници на виртуалния клас (т.е. класовете B и C), но не на по-далечните му наследници. Всички последни (крайни, т.е. листа) наследници, на каквато и степен на йерархия да се намират, наследяват по един екземпляр от виртуалния клас. Затова за производния клас D сега е достатъчен конструктор с четири параметри.
Модифицираната част от файла за реализация на класовете:

```
D::D(int k, int l, int m, int n):A(k),B(0,l), C(0,m), d(n) {}
void D::show()
{cout << " a=" << a << " b=" << b << " c=" << c << " d=" << d << endl;
}
```

Обърнете внимание на реализацията на конструктора. В списъка за инициализация присъства явно обръщение към конструктора A, а първите параметри на B и C не се използват (използвана е 0). Тук е показано следното правило за реализация на конструктори: всеки наследник на виртуален базов клас, на каквото и ниво на йерархия да се намира, трябва явно да извика конструктора на виртуалния базов клас, освен конструкторите на своите непосредствени родители. При това на последните е забранен достъпа до виртуалните данни(нулевите параметри в нашия случай няма да се използват). От друга страна, при създаването на обектите на непосредствените наследници на виртуалния клас (в нашия случай – B или C) съответните конструктори имат достъп до всички свои данни-елементи, в това число и до наследените виртуални.

Втората особеност на разглеждания програмен фрагмент е липсата на необходимост от използване на операцията за разрешаване на област на действие при обръщението към всички данни-елементи (в това число и към данната-елемент a), тъй като не възниква неоднозначност: всички те съществуват в класа D в единствен екземпляр.
Освен казаното, приведения по-долу текст на главната функция илюстрира възможността за използване на указатели за класове от всякакво ниво на йерархия за обръщение към обектите на производния клас D (припомняме предишния пример, където използването на указател от клас A беше забранено):

```
#include “abcd.h”
void main() {
A *a = new A(1);
a->show();
delete a;
a = new B(2,3); //  Конструкторът използва и двата параметъра
a->show();
delete a;
a = new C(4,5);
a->show();
delete a;
a = new D(6, 7, 8, 9); // Обръщение по указател на виртуалния
базов клас
a->show();
delete a;
B *b = new D(10, 11, 12, 13);
b->show();
delete b;
```

```
C *c = new D(14, 15, 16, 17);
c->show();
delete c;
}
```

Резултат:

```
a=1
a=2 b=3
a=4 c=5
a=6 b=7 c=8 d=9
a=10 b=11 c=12 d=13
a=14 b=15 c=16 d=17
```

33. Пример за използване на виртуални функции и полиморфизъм

В примера ще се пресмята заплата на служители, като се отчита тяхната длъжност. Базовият клас е Employee, производните класове са – клас Boss (с фиксирана заплата), клас CommissionWorker (с базова заплата и комисионен процент), клас PieceWorker (със заплата, пропорционална на изработеното количество), клас HourlyWorker (със заплата, пропорционална на часовете за работа).
Функцията earning се използва за всички служители, но начинът за пресмятане на заплата е различен, затова тя се декларира като виртуална в базовия клас, а след това е реализирана във всеки от производните класове. За да се пресметне заплатата, в програмата се използва указател от базовия клас, който сочи към обект на конкретен служител при извикване на функцията earning.

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

```
#include <string.h>
#include <assert.h>
#include <iomanip.h>
class Employee {
 public:
   Employee (const char *, const char *);
   ~Employee ();
   const char *getFirstName () const;
   const char *getLastName () const;
   virtual float earnings () const = 0; // чисто виртуална
   virtual void print () const = 0; // чисто виртуална
/* Защо тези функции са декларирани като чисто виртуални? Няма
смисъл да се реализират тези функции в класа Employee. Не може
да се начислява заплата на абстрактен служещ: най-напред ние
трябва да определим типа на служещия. Не може да се печата
заплата на абстрактен служещ. Правейки тези функции чисто
виртуални, ние показваме, че те трябва да бъдат реализирани в
производните класове, а не в базовия. */
```

```
 private:
   char *firstName;
   char *lastName;
} ;
Employee::Employee (const char *first, const char *last)
 { firstName = new char [strlen (first) + 1];
   assert (firstName != NULL);
   strcpy (firstName, first);
   lastName = new char [strlen (last) + 1];
   assert (lastName != NULL);
   strcpy (lastName, last);
 }
Employee::~Employee ()
 { delete []firstName;
   delete []lastName;
 }
const char *Employee::getFirstName () const
 { return firstName;}
```

```
const char *Employee::getLastName () const
 { return lastName;}
class Boss: public Employee {
 public:
   Boss ( const char *, const char *, float = 0);
   void setWeeklySalary (float);
   virtual float earnings () const;
   virtual void print () const;
 private:
   float weeklySalary;
} ;
Boss::Boss (const char *first, const char *last, float s) :
Employee (first, last)
 { setWeeklySalary (s); }
void Boss::setWeeklySalary (float s)
{ weeklySalary = s > 0 ? s : 0; }
float Boss::earnings () const
{ return weeklySalary; }
```

```
void Boss::print () const
{ cout << endl <<  “Администратор: “ << getFirstName () << ‘ ‘
        << getLastName ();
}
/* Класът Boss  е производен от класът Employee с открито
наследяване. Откритите функции-елементи включват: конструктор,
който приема като аргументи име, фамилия и седмична заплата,
и предава името и фамилията на конструктора на Employee за
инициализация на елементите firstName и lastName на тази част от
обекта на производния клас, която съвпада с базовия клас;
функция setWeeklySalary, която присвоява нова стойност на
скритата данна-елемент weeklySalary; виртуална функция earnings,
в която се определя, как да се начислява заплатата в класа Boss;
виртуална функция print, която извежда тип на длъжността на
служещия и неговото име. */
class CommissionWorker: public Employee {
 public:
   CommissionWorker ( const char *, const char *, float = 0, float = 0,
```

```
   unsigned = 0);
   void setSalary (float);
   void setCommission (float);
   void setQuantity (unsigned);
   virtual float earnings () const;
   virtual void print () const;
 private:
   float salary;
   float commission;
   unsigned quantity;
} ;
CommissionWorker:: CommissionWorker (const char *first, const char
*last, float s, float c, unsigned q) : Employee (first, last)
 { setSalary (s); setCommission (c); setQuantity (q); }
void CommissionWorker::setSalary (float s)
{ salary = s > 0 ? s : 0; }
void CommissionWorker::setCommission (float c)
{ commission = c > 0 ? c : 0; }
```

```
void CommissionWorker::setQuantity (unsigned q)
{ quantity = q > 0 ? q : 0; }
float CommissionWorker::earnings () const
{ return salary + commission*quantity; }
void CommissionWorker::print () const
{ cout << endl << “Служещ на комисионна: “ << getFirstName () << ‘ ‘
        << getLastName ();
}
```

/* Класът ComissionWorker е производен от класа Employee с открито наследяване. Откритите функции-елементи включват: конструктор, който има за аргументи име, фамилия, базова седмична заплата, комисионно възнаграждение и брой продадени изделия, а също предава името и фамилията на конструктора на Employee; функции set, които присвояват нови стойности на скритите данни-елементи salary, commission и quantity; виртуална функция earnings, в която се определя как се пресмята заплатата в класа ComissionWorker; виртуална функция print, която извежда длъжността на служещия и неговото име. */

```
class PieceWorker: public Employee {
 public:
   PieceWorker ( const char *, const char *, float = 0, unsigned = 0);
   void setWagePerPiece (float);
   void setQuantity (unsigned);
   virtual float earnings () const;
   virtual void print () const;
 private:
   float wagePerPiece;
   unsigned quantity;
} ;
PieceWorker:: PieceWorker (const char *first, const char *last, float w,
unsigned q) : Employee (first, last)
 { setWagePerPiece (w); setQuantity (q); }
void PieceWorker::setWagePerPiece (float w)
{ wagePerPiece = w > 0 ? w : 0; }
void PieceWorker::setQuantity (unsigned q)
{ quantity = q > 0 ? q : 0; }
```

```
float PieceWorker::earnings () const
{ return wagePerPiece*quantity; }
void PieceWorker::print () const
{ cout << endl << “Заплата на парче: “ << getFirstName () << ‘ ‘
        << getLastName ();
}
```
/* Класът PieceWorker е производен от класа Employee с открито наследяване. Откритите функции-елементи включват: конструктор, който има за аргументи име, фамилия, заплащане за единица продукция и количество произведена продукция за седмица, а също предава името и фамилията на конструктора на Employee; функции set, които присвояват нови стойности на скритите данни-елементи wagePerPiece и quantity; виртуална функция earnings, в която се пресмята заплатата в класа PieceWorker; виртуална функция print, която извежда длъжността на служещия и неговото име. */
```
class HourlyWorker: public Employee {
 public:
```

```
   HourlyWorker ( const char *, const char *, float = 0, float = 0);
   void setWage (float);
   void setHours (float);
   virtual float earnings () const;
   virtual void print () const;
 private:
   float wage;
   float hours;
} ;
HourlyWorker:: HourlyWorker (const char *first, const char *last, float w,
float h) : Employee (first, last)
 { setWage (w); setHours (h); }
void HourlyWorker::setWage (float w)
{ wage = w > 0 ? w : 0; }
void HourlyWorker::setHours (float h)
{ hours = h > 0 && h <= 168 ? h : 0; }
float HourlyWorker::earnings () const
{ return hours < 40 ? wage*hours: 40*wage + 1.5*(hours-40)*wage; }
```

```
void HourlyWorker::print () const
{ cout << endl << “Почасова заплата: “ << getFirstName () << ‘ ‘
        << getLastName ();
}

/* Класът HourlyWorker е производен от класа Employee с открито
наследяване. Откритите функции-елементи включват: конструктор,
който има за аргументи име, фамилия, почасова заплата и
количество отработени часове, а също предава името и
фамилията на конструктора на Employee; функции set, които
присвояват нови стойности на скритите данни-елементи wage и
hours; виртуална функция earnings, в която се определя как се
пресмята заплатата в класа HourlyWorker; виртуална функция print,
която извежда длъжността на служещия и неговото име. */
void main ()
/* Програмата драйвер започва с дефиниране на указател ptr от
базовия клас от тип Employee *. Трите сегмента от кода във
```

функцията main са сходни един с друг, затова ще обсъдим само първия сегмент, който е свързан с обекта Boss. */

```
{
 cout << setiosflags (ios::showpoint) << setprecision (2);
 Employee *ptr;
 Boss b (“Иван”, “Христов”, 800);
 ptr = &b;
 ptr -> print ();//динамично свързване
 cout << “Изработил  $” << ptr -> earnings () << endl;//динамично
свързване
 b.print ();//статично свързване
 cout << “Изработил $” << b.earnings () << endl; //статично
свързване
 CommissionWorker c (“Елена”, “Кирова”, 200, 3, 150);
 ptr = &c;
 ptr -> print (); // динамично свързване
```

```
 cout << “Изработил $” << ptr -> earnings () << endl; // динамично
свързване
 c.print (); //статично свързване
 cout << “Изработил $” << c.earnings () << endl; //статично
свързване
 PieceWorker p (“Калоян”, “Николов”, 2.5, 200);
 ptr = &p;
 ptr -> print (); // динамично свързване
 cout << “Изработил $” << ptr -> earnings () << endl; // динамично
свързване
 p.print (); //статично свързване
 cout << “ $” << p.earnings () << endl; //статично свързване
 HourlyWorker h (“Първан”, “Първанов”, 13.75, 40);
 ptr = &h;
 ptr -> print ();// динамично свързване
 cout << “Изработил $” << ptr -> earnings () << endl; // динамично
свързване
 h.print (); //статично свързване
```

```
 cout << “Изработил $” << h.earnings () << endl; //статично
свързване
}
```
Резултат:
```
Администратор: Иван Христов Изработил $800.00
Администратор: Иван Христов Изработил $800.00
Служещ на комисионна: Елена Кирова Изработил $650.00
Служещ на комисионна: Елена Кирова Изработил $650.00
Заплата на парче: Калоян Николов Изработил $500.00
Заплата на парче: Калоян Николов Изработил $500.00
Почасова заплата: Първан Първанов Изработил $550.00
Почасова заплата: Първан Първанов Изработил $550.00
```
Редът

`Boss b (“Иван”, “Христов”, 800);`

създава обект b от производния клас Boss и предава на конструктора аргументи, включващи име, фамилия и фиксирана седмична заплата.

Редът

ptr = &b;
зарежда указателя от базовия клас ptr с адреса на обекта от производния клас.
Това трябва да направим за реализация на полиморфното поведение.

Редът
ptr->print( ); // динамично свързване
извиква функцията-елемент print на този обект, към когото сочи ptr. Тъй като функцията print е обявена в базовия клас като виртуална, системата извиква функцията print на обекта от производния клас както трябва при полиморфно поведение. Това обръщение към функцията е пример за динамично свързване: функцията се извиква чрез указателя от базовия клас, затова избора каква функция да се извика, се отлага до времето за изпълнение на програмата.
Редът

cout << " Изработил $" <<ptr->earnings(); // динамично свързване извиква функцията-елемент earnings от този обект, към който сочи ptr. Тъй като функцията earnings е обявена в базовия клас като виртуална, системата извиква функцията earnings на обекта от производния клас. Това също е пример на динамично свързване.

Редът
b.print(); // статично свързване
явно извиква функцията-елемент print от класа Boss, използвайки операцията за достъп до елемент '.' от зададения обект b от класа Boss. Това е пример за статично свързване, тъй като типът на обекта, за който се извиква функцията, е известен по време на компилацията. Това извикване е включено с цел сравнение, за да се покаже, правилната функция print ли се извиква при използване на динамичното свързване.
Редът
cout << " Изработил $" << b.earnings(); // статично свързване

явно извиква функцията-елемент earnings от класа Boss с помощта на операцията за достъп до елемент ‘.’ от зададения обект b от класа Boss. Това също е пример на статично свързване. Това извикване е включено с цел сравнение, за да се покаже, правилната функция earnings ли се извиква при използване на динамичното свързване.
В базовия клас Employee функциите getFirstName и getLastName връщат константни указатели към скритите данни-елементи, което не позволява на клиентите да модифицират тези данни. За да се избегне получаването на неопределен указател, клиентът трябва да копира получените низове, преди те да бъдат унищожени от деструктора на класа.

34. Пример за наследяване на интерфейса и неговата реализация

Ще разгледаме йерархията на фигурите Point, Circle и Cylinder. Допълнително във върха на йерархията ще създадем абстрактен базов клас Shape, който съдържа четири виртуални функции – две чисто виртуални за отпечатване на имената на фигурите и техните характеристики и две виртуални за лице и обем, реализирани да връщат стойност 0. Например класът Point не предефинира функциите за лице и обем, тъй като те имат стойност 0 за точка. Ще отбележим, че макар клас Shape да е абстрактен базов клас, той съдържа реализацията на някои функции и тези реализации се наследяват. Класът Shape предоставя за наследяване интерфейс във вид на четири виртуални функции, които ще влизат на всички нива в йерархията. В класа Shape са представени също някои реализации, които ще се използват от производните класове.
В примера се акцентира върху това, че класовете могат да наследяват и интерфейса, и реализацията на базовия клас. В йерархия, проектирана за наследяване на реализация, стремежът е да се осигури функционална възможност на възможно по-високо ниво за да може всеки нов производен клас да наследява

функции-елементи, описани в базовия клас и да използва тези описания.
В йерархия, проектирана за наследяване на интерфейс, стремежът е да се осигури функционална възможност на възможно по-ниско ниво. В този случай базовият клас декларира функции, които трябва да се извикват идентично за всеки обект в йерархията, т.е. извикванията да имат еднаква сигнатура. Производните класове сами осигуряват реализации на тези функции

```
#include <iostream.h>
#include <iomanip.h>
class Shape {
 public:
   virtual float area () const { return 0; }
   virtual float volume () const { return 0; }
   virtual void printShapeName () const = 0; // чисто виртуална
   virtual void print () const = 0; // чисто виртуална
} ;
```

```
class Point : public Shape {
 friend ostream &operator << (ostream &, const Point &);
 public:
   Point (float = 0, float = 0);
   void setPoint (float, float);
   float getX () const { return x; }
   float getY () const { return y; }
   virtual void printShapeName () const { cout << “Точка: “; }
   virtual void print () const;
 private:
   float x, y;
} ;
Point::Point (float a, float b)
 { setPoint (a, b); }
void Point::setPoint (float a, float b)
 { x = a; y = b; }
void Point::print () const
 { cout << ‘[‘ << x << “, “ << y << ‘]’; }
```

```
ostream &operator<< (ostream &output, const Point &p)
 { p.print();  return output; }
/* Класът Point  е производен от класа Shape с открито
наследяване. Точката няма нито лице, нито обем, така че
функциите-елементи на базовия клас area и volume в дадения
случай не са предефинирани. Те наследяват техните дефиниции
от базовия клас Shape.
Функциите printShapeName и print са реализации на виртуални
функции, които са декларирани в базовия клас като чисто
виртуални. Други функции-елементи са функцията set,
присвояваща нови стойности на координатите на точката x и y, и
функция get, връщаща координатите x и y. */
class Circle : public Point {
 friend ostream &operator<< (ostream &, const Circle &);
 public:
   Circle (float r = 0, float x = 0, float y = 0);
   void setRadius (float);
   float getRadius () const;
```

```
   virtual float area () const;
   virtual void printShapeName () const
    { cout << “Кръг: “; }
   virtual void print () const;
 private:
   float radius;
} ;
Circle::Circle (float r, float a, float b) : Point (a, b)
{ radius = r > 0 ? r : 0; }
void Circle::setRadius (float r)
 { radius  = r > 0 ? r : 0; }
float Circle::getRadius () const
 { return radius; }
float Circle::area () const
 { return 3.14159*radius*radius; }
void Circle::print () const
 { cout << ‘[‘ << getX() << “, “ << getY() << “]; Радиус = “ <<
setiosflags(ios::showpoint)
```

```
          << setprecision (2 ) << radius; }
ostream &operator<< (ostream &output, const Circle &c)
 {  c.print();
    return output; }
```
/* Класът Circle е производен от класа Point с открито наследяване. Кръгът няма обем, така че функцията-елемент на базовия клас volume за класа Circle не се предефинира.
Тя се наследява от базовия клас Shape (през класа Point). Кръгът има лице, затова функцията area в класа Circle се предефинира. Функциите printShapeName и print са реализации на виртуални функции, които са дефинирани в базовия клас като чисто виртуални. Ако тези функции не се предефинират тук, то щяха да наследяват версиите на тези функции от класа Point. Други функции-елементи са функцията set, присвояваща нова стойност на радиуса radius, и функцията get, връщаща стойността на радиуса. */

```
class Cylinder : public Circle {
 friend ostream &operator<< (ostream &, const Cylinder &);
```

```
 public:
  Cylinder (float h = 0, float r = 0, float x = 0, float y = 0);
  void setHeight(float);
  virtual float area () const ;
  virtual float volume () const;
  virtual void printShapeName () const
   { cout << “Цилиндър: “; }
  virtual void print () const;
 private:
  float height;
} ;
Cylinder::Cylinder (float h, float r, float a, float b) : Circle (r, a, b)
 { height = h > 0 ? h : 0; }
void Cylinder::setHeight(float h) { height = h > 0 ? h : 0; }
float Cylinder::area() const
{
return 2 * Circle::area() + 2 * 3.14159 * Circle::getRadius() * height;}
float Cylinder::volume() const { return Circle::area() * height; }
```

```
void Cylinder::print () const
 { Circle::print ();
   cout << “; Височината = “ << height;
 }
ostream &operator<<(ostream &output, const Cylinder &c)
 { c.print ();
   return output;
 }
```
/*Класът Cylinder е производен от класа Circle с открито наследяване. Цилиндрите имат лице и обем, затова в класа Cylinder двете функции area и volume са предефинирани. Функциите printShapeName и print са реализации на виртуални функции, които са декларирани в базовия клас като чисто виртуални. Ако тези функции не се предефинираха тук, то щяха да наследяват версиите на тези функции от класа Circle. Други функции-елементи са функцията set, присвояваща нова стойност на височината на цилиндъра height, и функцията get, връщаща стойността на височината на цилиндъра. */

void main ()
 { cout << setprecision (2) << setiosflags (ios::fixed | ios::showpoint);
/* Програмата драйвер започва работата си със създаване на обект point от класа Point, обект circle от класа Circle и обект cylinder от класа Cylinder.

За всеки обект се извиква функцията printShapeName и всеки обект се извежда с помощта на своята предефинирана операция включване в поток, за да покаже правилната инициализация на обектите. След това се дефинира масив array от тип Shape *. Този масив от указатели от базовия клас се използва за да сочи към всеки създаден обект от производния клас. Отначало на елемента от масива array[0] се присвоява адреса на обекта point, после на елемента array[1] се присвоява адреса на обекта circle, а на елемента array[2] се присвоява адреса на обекта cylinder. След това се изпълнява цикъл for, преглеждащ масива array и изпълняващ на всяка итерация на цикъла следните обръщения:
array [i]->printShapeName;

```
array [i]->print( )
array [i]->area( )
array [i]->volume( ) */
   Point point (7, 11); // създава се обект на класа Point
   Circle circle (3.5, 22, 8); // създава се обект на класа Circle
   Cylinder cylinder (10, 3.3, 10, 10); // създава се обект на класа
Cylinder
   point.printShapeName (); // статично свързване
   cout << point << endl;
   circle.printShapeName (); // статично свързване
   cout << circle << endl;
   cylinder.printShapeName (); // статично свързване
   cout << cylinder << endl;
   Shape *array[3];
   array [0] = &point; array[1] = &circle; array[2] = &cylinder;
// Цикъл по array и печат на името на фигурата, лицето и обема на
всеки
```

```
// обект, към който сочи елемент (използва се динамично
свързване)
   for (int i = 0; i < 3; i++) {
    array[i] -> printShapeName ();
    cout << endl;
    array[i] -> print ();
    cout << “\nЛице = “ << array[i] -> area ()
         << “\nОбем = “ << array[i] -> volume () << endl;
    }
```

/* Всяко от показаните по-горе обръщения активизира тези функции за този обект, към който сочи елемента от масива array [i]. От резултите следва, че функциите се извикват както трябва. Отначало се извежда ред "Точка:" и координатите, съхраняващи се в обекта point; лицето и обема са равни на 0.00.
След това се извежда ред "Кръг:", координатите на центъра на кръга и радиуса, съхраняващи се в обекта circle. След това се изчислява лицето на кръга, но неговия обем е равен на 0.00. И, накрая, се извежда ред "Цилиндър", координатите на центъра,

радиуса и височината на цилиндъра, съхраняващи се в обекта cylinder, след това се изчислява лицето му и обема. Всичките извиквания на функциите printShapeName, print, area и volume се реализират в процеса на изпълнение на програмата с помощта на динамично свързване. */
}

Резултат:
Точка: [7, 11]
Кръг: [22, 8]; Радиус =3.50
Цилиндър: [10.00, 10.00]; Радиус = 3.30; Височина = 10.00
Точка: [7.00, 11.00] Лице = 0.00 Обем =0.00
Кръг: [22.00, 8.00]; Радиус =3.50 Лице = 38.48 Обем =0.00
Цилиндър: [10.00, 10.00]; Радиус = 3.30; Височина = 10.00 Лице = 275.77
Обем = 342.12