15. Обектно ориентирано програмиране. Основни принципи. Класове и обекти. Наследяване и инкапсулация. Параметричен полиморфизъм.

Обекти (екземпляри). Класове. Декларация на клас и декларация на обект. Конструктори – конструктор по подразбиране, конструктор за присвояване. Освобождаване на памет – процедура за събиране на свободната памет. Методи – декларация, предаване на параметри, връщане на резултат. Наследяване и достъп до наследените компоненти. Еднократно и многократно (множествено) наследяване. Производни и вложени класове. Абстрактни методи и класове. Инкапсулация и скриване на информацията. Модификатори – статични полета и методи. Параметричен полиморфизъм. Типовете като параметри към функция и клас.

Структури от данни –> класове – пример с рационални числа

При подхода **абстракция със структури данни** методите за използване на данните са разделени от методите за тяхното представяне. Програмите се конструират така, че да работят с абстрактни данни – данни с неуточнено представяне. След това представянето се конкретизира с помощта на множество функции: **конструктори**, **селектори** и **мутатори**, които реализират абстрактните данни по конкретен начин.
За да илюстрираме подхода ще създадем програма за рационално-числова аритметика. В програмата се дефинират функции за събиране, изваждане, умножение и деление на рационални числа по общоизвестните правила. Тези операции лесно могат да се реализират, ако има начин за конструиране на рационално число по зададени две цели числа, представящи съответно неговия числител и знаменател и ако има начини за извличане на числителя и знаменателя на рационално число. Затова в програмата се дефинират следните функции:

- void makerat(rat &r, int a, int b) – конструира рационално число r по зададени числител a и знаменател b;
- int numer (rat &r) – извлича числителят на рационалното число r; (numerator)
- int denom (rat &r) – извлича знаменателят на рационалното число r. (denumerator)

В тези декларации rat е името на типа рационално число. Все още не знаем как са дефинирани трите функции, но ако предположим, че те са реализирани, то функциите за рационално-числова аритметика и процедурата за извеждане на рационално число могат лесно да се реализират. Например, функцията за събиране се реализира така:

```
rat sumrat (rat &r1, rat &r2){
        rat r;
        makerat (r, numer (r1) * denom (r2) + numer (r2) * denom (r1), denom (r1) * denom (r2));
        return r;
}
```

Останалите операции subrat, multrat, quotrat за изваждане, умножение и деление на рационални числа се реализират аналогично. Функцията за извеждане се дефинира така:

```
void printrat (rat &r){
        cout << numer (r) << '/' << denom (r) << endl;
}
```

Сега ще се върнем към представянето на рационалните числа, а също към реализация на примитивните операции: конструктора makerat и селекторите numer и denom. Тъй като рационалните числа се представят чрез наредена двойка от цели числа, удобно е използването на структура:

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

Тогава примитивните функции, които реализират конструктора и селекторите имат вида:

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

След като са дефинирани всички функции можем да имаме следният програмен фрагмент в някоя функция:

```
…
rat r1, r2, r3;
makerat (r1, 1, 2);
makerat (r2, 3, 4);
r3 = sumrat (r1, r2);
printrat (r3);
…
```

След изпълнението на този фрагмент ще се изведе 10/8.

В примера се вижда, че реализирането на подхода абстракция със структури данни има четири нива на абстракция:

- използване на рационалните числа в проблемната област (намиране на сумата на 1/2 и 3/4);
- реализиране на правилата за рационално-числова аритметика (sumrat, subrat, multrat, quotrat, printrat);
- избор на представяне за рационалните числа и реализиране на примитивни конструктори и селектори (makerat, numer, denom);
- работа на ниво структура.

Използването на подхода абстракция със структури данни прави програмите по-лесни за описание и модификация.

Недостатък на горната реализация е, че функциите не съкращават рационалните числа. За да се поправи този недостатък, обаче, трябва да се промени единствено функцията makerat. Да предположим, че функцията gcd намира най-големия общ делител на две естествени числа. Тогава новата makerat има вида:

```
void makerat (rat &r, int a, int b)
{
        if (a == 0) { r.num = 0; r.den = 1; }
        else {
                int aa = (a > 0) ? a : -a;
                int ab = (b > 0) ? b : -b;
                int g = gcd (aa, ab);
                if ((a > 0 && b > 0) || (a < 0 && b < 0)){
                r.num = aa/g; r.den = ab/g; }
                else { r.num = – aa/g; r.den = ab/g; }
                }
}
```

С това проблемът със съкращаването на рационални числа е решен. За неговото решаване се наложи малка модификация, която засегна само конструктора makerat. Така илюстрирахме лесната модифицируемост на програмите, реализиращи подхода абстракция със структури данни.

Изрично ще отбележим, че дефинираната структура rat, която представя рационално число не може да се използва като тип данни рационално число. Това е така, защото при типовете данни представянето на данните е скрито за потребителя.

За всеки тип данни е известно множество от допустимите стойности и множество от вградените функции и операции, допустими за този тип. Така възниква усещането, че представянето на рационално число като запис с две полета трябва да се обедини с примитивните операции (makerat, numer, denom). Последното е възможно, тъй като в C++ се допуска полетата на структура да бъдат функции. За да направим това в нашия пример извършваме следните стъпки:

1. Включваме декларациите на makerat, numer и denom в дефиницията на структурата rat, като елиминираме формалният параметър r и в трите функции.

Това води до следната дефиниция на структурата rat:

```
struct rat {
        int num;
        int den;
        void makerat (int a, int b);
        int numer();
        int denom();
};
```

2. Отразяваме промените в дефинициите на функциите makerat, numer и denom – премахваме формалният параметър r и пред името на всяка от тези функции поставяме името на структурата rat и операцията за разрешаване на достъп ::.

Така получаваме следните дефиниции:

```
void rat::makerat (int a, int b){
        if (a == 0) { num = 0; den = 1; }
        else {
                int aa = (a > 0) ? a : -a;
                int ab = (b > 0) ? b : -b;
                int g = gcd (aa, ab);
                if ((a > 0 && b > 0) || (a < 0 && b < 0)){ num = aa/g; den = ab/g; }
                else { num = – aa/g; den = ab/g; }
                }
}
```

```
int rat::numer () { return num; }
int rat::denom (rat r) { return den; }
```

Функциите makerat, numer, denom при тази модификация се наричат **член-функции** на структурата rat. Извикването им се осъществява като полета на структура, например (r е променлива от тип rat) r.makerat(1, 5), r.denom(), r.numer().
Сега забелязваме, че във функциите sumrat, subrat, multrat, quotrat и printrat не се използват полетата на записа num и den, но ако направим опит за използването им даже на ниво main, той ще бъде успешен. Последното може да се забрани, ако се използва етикетите private: пред дефиницията на полетата num и den и public: пред декларациите на член-функциите.
Структурата rat приема вида:

```
struct rat {
 private:
        int num;
        int den;
public:
        void makerat (int a, int b);
        int numer();
        int denom();
};
```

Опитът за използване на полетата num и den на структурата rat извън нейните член-функции води до грешка.
Сега ако заменим запазената дума struct със запазената дума class и запазим всички останали дефиниции получаваме еквивалентна програма. Единствената разлика е, че достъпът по подразбиране вътре в дефиницията в този случай е private, а не public.
Така естествено достигнахме до понятието **клас.**

Класовете са типове данни, дефинирани от потребителя. Те могат да обогатяват възможностите на вече съществуващ потребителски тип или да представят нов тип данни. Както вече видяхме, класовете са подобни на структурите и в повечето отношения са идентични – всеки клас може да се разгледа като структура, на която са наложени някои ограничения по отношение на правата на достъп. Всеки клас съдържа данни, наречени **данни-елементи** (**член-данни**) на класа и набор от функции, наречени **функции-елементи** (**член-функции**), които обработват данните-елементи на класа. Понякога функциите-елементи в обектно-ориентираното програмиране се наричат **методи.**

Модификатори – статични полета и методи

Дефинирането на един клас се състои от две части:

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

Декларацията на клас се състои от заглавие и тяло. Заглавието започва със запазената дума class, последвана от името на класа. Тялото е заградено във фигурни скоби. След тези скоби стои списък от имена на обекти (може да е празен), разделени със запетаи и накрая се записва ‘;’. В тялото на класа са декларирани членовете на класа (член-данни и член-функции). Имената на членовете на класа са локални за този клас, т.е. в различните класове в рамките на една програма могат да се дефинират членове с еднакви имена. Член-данни от един и същи тип могат да се изредят, разделени със запетаи и предшествани от съответния тип. Изрично ще отбележим, че типът на член-данна на един клас не може да съвпада с името на този клас, но това не важи за типа на член-функциите и типовете на техните параметри. В тялото някои декларации могат да бъдат предшествани от **спецификаторите за достъп public, private, protected.** Областта на един спецификатор за достъп започва от самия него и продължава до следващия спецификатор или до края на декларацията на класа, ако няма следващ спецификатор. Подразбиращ се спецификатор за достъп е private. Един и същ спецификатор за достъп може да се използва повече от един път в декларацията на клас.
Достъпът до членовете на класовете може да се разглежда на следните две нива:

- ниво член-функции;
- ниво външни функции.

Член-функциите на един клас имат достъп до всички членове на този клас. При това не е необходимо тези компоненти да се предават като параметри. Този режим на достъп се нарича **режим на пряк достъп**. Режимът на достъп до членовете на класа за външните функции се определя от начина на дефиниране на членовете. Членовете на един клас, декларирани като private са видими само в рамките на класа и външните функции нямат достъп до тях. Чрез използване на членове на класа, декларирани като private, се постига скриването на тези членове за външната за класа среда. Този процес на скриване се нарича още **капсулиране на информацията**. Членовете на клас, които трябва да бъдат видими извън класа (т.е. достъпни за външни функции) трябва да бъдат декларирани като public. Освен като private и public, членовете на класа могат да бъдат декларирани като protected – тогава те имат същото поведение като private, но разликата е, че те могат да бъдат достъпни от производните класове на дадения клас.
След като един клас е деклариран, трябва да се дефинират неговите методи. Член-данните на един клас се дефинират в рамките на тялото на класа. За член-функциите има два варианта: дефинират се в тялото на класа или извън тялото на класа, като във втория случай в тялото на класа се посочват само техните прототипи. Когато една член-функция се дефинира извън

тялото на съответния клас, нейното име трябва да се предшества от името на класа, към който принадлежи член-функцията, последвано от бинарната операция за разрешаване на достъп.
В случай, че една член-функция на клас е дефинирана вътре в тялото на класа, тя се разглежда като **вградена функция** (inline).
Ще отбележим, че в тялото на дефиницията на член-функция явно не се използва обектът, върху който тя ще се приложи. Той участва неявно – чрез член-данните на класа. Поради това той се нарича **неявен** параметър на член-функцията. Останалите параметри, които участват явно в дефиницията на член-функцията се наричат **явни**. Всяка член-функция има точно един неявен параметър и нула или повече явни.

Член-данните и член-функциите на класа имат **област на действие клас**. Външните функции, 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, елементите на класа, които са декларирани като public, могат да се използват или с помощта на обект от този клас, или чрез псевдоним на обект от този клас, или чрез указател към обект от този клас. Компонентите на функциите-елементи имат област на действия функция – променливите, които са дефинирани в тялото на функцията-елемент са известни само на тази функция. Ако във функция-елемент се дефинира променлива с име, съвпадащо с името на член-данна от област на действие клас, то първата скрива втората. Такива скрити променливи могат да станат достъпни с помощта на бинарната операция за разрешаване на достъп с ляв операнд името на класа и десен операнд името на съответната член-данна. За разлика от функциите, класовете могат да се дефинират на различни нива в програмата: **глобално** (в област на действие файл) и **локално** (вътре в тялото на функция или в тялото на клас). Областта на глобално деклариран клас започва от декларацията на класа и продължава до края на програмата. Ако клас е дефиниран в рамките на тялото на функция, всички негови член-функции трябва да са вградени, тъй като в противен случай би се получило недопустимото за C++ влагане на функции. Областта на клас, дефиниран във функция започва от мястото, където той е деклариран и завършва в края на блока, от който е част тази декларация. Обекти на такъв клас могат да се дефинират и използват само във въпросния блок в тялото на функцията.

Обект

След като един клас е дефиниран, могат да се създават негови екземпляри, наречени **обекти**. Връзката между клас и обект в C++ е подобна на връзката между тип данни и променлива, но за разлика от обикновените променливи, обектите се състоят от множество компоненти (член-данни и член-функции). При всяко дефиниране на обект на класа автоматично се извиква точно един от конструкторите на класа с цел да се инициализира обекта. Ако дефиницията е без явна инициализация, дефинираният обект се инициализира чрез конструктора по подразбиране. Явна инициализация може да се извърши по два начина:

- името на обекта в дефиницията е последвано от скоби, в които е поместен списък от начални стойности – тези стойности се предават като параметри в подходящ конструктор на класа;
- името на обекта е последвано от знак за равенство и друг, вече дефиниран обект от класа.

При компилиране на дефиницията на клас не се заделя никаква памет. Памет се заделя едва при дефиниране на конкретни обекти от този клас. Достъпът до компонентите на обекта (ако е възможен), се осъществява по два начина – пряко и косвено. При прекия достъп се използва името на обекта и името на данната или метода, разделени с '.'. При косвения достъп се използва името на указател към обекта и името на данната или метода, разделени с '->'. Двата метода за достъп са взаимнозаменяеми: x.a ⇔ (&x) -> a, px -> a ⇔ (*px).a.
При създаването на обекти на един клас, кодът на методите на този клас не се копира за всеки обект, а се намира само на едно място в паметта. Естествено, възниква въпросът по какъв начин методите на един клас разбират за кой обект на този клас са били извикани. Отговорът на този въпрос дава **указателят this.** Всяка член-функция на клас поддържа допълнителен неявен формален параметър указател this от тип <име_на_клас>*. Това става по следния начин: компилаторът преобразува всяка член-функция на клас до обикновена функция с уникално име и допълнителен параметър указателят this и вътре в тялото на тази функция преките обръщения към членове на класа се преобразуват към косвени обръщения чрез this, съответно, след това, компилаторът преобразува всяко обръщение към член-функция на класа в програмата до обръщение към съответната обикновена функция, като за фактически параметър, отговарящ на формалния параметър this се използва адресът на обекта, за който е била извикана член-функцията.

Конструктор (constructor) – default constructor and copy constructor

Създаването на обекти е свързано със заделяне на памет, запомняне на текущо състояние, задаване на начални стойности и други дейности, които се наричат **инициализация** на обекта. В езика C++ тези действия се изпълняват от специален вид член-функции на класовете – **конструкторите**. Конструкторът е член-функция, която се отличава от останалите член-функции по следните характеристики:

- името на конструктора съвпада с името на класа;
- типът на резултата на конструктора не се указва явно в неговата декларацията – винаги се подразбира, че е типът на указателя this;
- конструкторът се изпълнява автоматично при създаване на обекти;
- конструкторът не може да се извиква явно чрез обект или косвено чрез указател към обект.

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

В заглавието на един конструктор при неговата дефиниция, непосредствено след списъка с формалните параметри може да има инициализиращ списък. Чрез този списък член-данни на класа могат да се свържат с начални стойности. Инициализиращият списък е съставен от имена на член-данни на класа, последвани от скоби, в които е записана началната стойност на съответната член-данна, като отделните елементи на списъка се разделят със запетаи и в началото му се записва символът ':'. В конструктора една член-данна може да се инициализира или чрез инициализиращия списък или вътре в тялото на конструктора. По обясними причини за константните член-данни е възможен само първият начин за инициализация.
В рамките на една програма може да се извършва **предефиниране** на функции, което означава използване на функции с еднакви имена в една и съща област на видимост. При осъществяване на извикване към предефинирана функция, компилаторът търси варианта на функцията с възможно най-добро съвпадение. Като критерии за добро съвпадане имаме следните нива на съответствие: точно съответствие (по брой и тип на формалните и фактическите параметри) и съответствие чрез разширяване на типа на някои от параметрите. В един клас може да се дефинират няколко конструктора. Всички те имат едно и също име (името на класа), така че трябва да се различават по броя и/или типа на своите параметри. При създаването на един обект се изпълнява точно един от предефинираните конструктори. Кой конкретен конструктор да се изпълни се определя съгласно критерия за най-добро съвпадение.
В един клас може явно да е дефиниран, но може и да не е дефиниран конструктор. Във втория случай автоматично се създава т.н. **конструктор по подразбиране**. Този конструктор реализира множество от действия като: заделяне на памет за данните на обект, инициализиране на някои системни променливи и др. В първия случай, т.е. когато за класа има дефиниран поне един конструктор, тогава конструктор по подразбиране не се създава автоматично и за да може да се дефинират обекти без да се инициализират, потребителят сам трябва да дефинира подразбиращ се конструктор – той няма параметри или всичките му параметри са подразбиращи се.
Инициализацията на новосъздаден обект от даден клас може да зависи от друг обект на същия клас. За да се укаже такава зависимост се използва знакът за присвояване или скоби, в които се записва името на обекта-инициализатор. Самата инициализация в този случай се извършва от специален конструктор, наречен **конструктор за присвояване** (**конструктор за копие**). Този конструктор поддържа точно един параметър от тип const <име_на_клас> &. Ако в един клас явно не е дефиниран конструктор за присвояване, компилаторът автоматично създава такъв, в първия момент когато се наложи новосъздаден обект от класа да се инициализира с обект от същия клас. Освен при инициализация чрез обект, конструкторът за присвояване се използва и при предаване по стойност на обект като аргумент на функция, а също и при връщане на обект като резултат от изпълнение на функция. Изрично ще отбележим, че конструкторът за присвояване се използва само в описаните случаи, но не и при присвояване на един обект на друг извън инициализация. В последния случай се използва предефинираната операция за присвояване.
Указателите към обекти се дефинират по същия начин както указател към стандартен тип данни. Например, ако в програмата е дефиниран класът MyClass, то са валидни следните дефиниции:

```
MyClass obj;
MyClass *ref = &obj;
```

Връзката между масиви и указатели е в сила и в случая, когато елементите на масива са обекти.
Елементите на един масив могат да са обекти, но разбира се от един и същи клас. Нека например е дефиниран класът MyClass, който има открита член-данна x и открита член-функция f без параметри.
Следната дефиниция е валидна:

```
MyClass obj[20];
```

Достъпът до елементите на масива е пряк и се осъществява по стандартния начин чрез индексиране: obj[0], obj[1], …, obj[19]. Тъй като obj[i], i = 0, 1, …, 19, е обект, към него са възможни обръщенията obj[i].x и obj[i].f(). Член-данна на един клас може да е масив. Например, да предположим, че MyClass има допълнителна открита член-данна y[5]. Нека obj е обект от клас MyClass. Тогава достъпът до елементите на масива y в обекта obj ще се осъществи по следния начин: obj.y[0], obj.y[1], …, obj.y[4]. Конструкторът по подразбиране играе важна роля при дефинирането и инициализирането на масиви от обекти. Масив от обекти, дефиниран в програмата се инициализира по два начина: явно (чрез инициализиращ списък) или неявно (чрез извикване на конструктора по подразбиране за всеки обект – елемент на масива).

Заделяне и освобождаване на динамична памет – new and delete

Всяка програма има няколко места за съхраняване на данни, едно от тях е **областта за динамични данни (heap)**. Динамичните данни са такива, че техният брой и размер не е известен в момента на проектиране на програмата. Те се създават и разрушават по време на изпълнение на програмата. След разрушаването им, заетата от тях памет се освобождава и може да се използва наново. Така паметта се използва по-ефективно.
Създаването и разрушаването на динамични данни в езика C++ се извършва чрез операцията new и оператора delete. Извикването на new заделя необходимата памет в heap-а и връща указател към нея. Извикването на delete освобождава паметта в heap-а, сочена от указател. На всяко извикване на new трябва да съответства извикване на delete, тъй като heap-ът не се чисти автоматично и в противен случай паметта рано или късно ще свърши. Синтаксисът на операцията new е следния:
new <име_на_тип> [[<размер>]] или
new <име_на_тип>(<инициализация>)
Тук <име_на_тип> е име на някой от стандартните типове или е име на клас, <размер> е произволен израз, който може да се преобразува до цял и показва броя на компонентите от тип <име_на_тип>, за които да се задели памет в heap-а,

<инициализация> е израз от тип <име_на_тип> или инициализация на обект според синтактиса на конструктор на класа, ако <име_на_тип> е име на клас. При това, <инициализация> не може да присъства, ако е зададен <размер>. Семантиката на операцията new е следната: в heap-а се заделят sizeof (<име_на_тип>) или sizeof (<име_на_тип>) * <размер> байта в зависимост от това дали не е указан или е указан <размер> и се инициализират чрез <инициализация>, ако такава има. Операцията връща като резултат указател към (първия елемент от) заделената памет. Ако в heap-а няма достатъчно памет, new връща NULL.
Синтаксисът на оператора delete е следния: **delete** <указател_към_динамичен_обект>.
Тук <указател_към_динамичен_обект> е указател към динамична данна, създадена чрез new. Семантиката на delete е следната: разрушава данната, адресирана от указателя и паметта, заемана от тази данна се освобождава. Ако данната, адресирана от указателя е обект на клас преди да се разруши се извиква деструктора на класа. Ако динамичната данна е масив (в заделянето на памет за нея с new е бил указан <размер> по-голям от 1), то delete трябва да се използва в следната форма: delete [] <указател>.

Деструктор (destructor)

Разрушаването на обекти на класовете в някои случаи е свързано с извършването на определени действия, които се наричат заключителни. Най-често тези действия са свързани с освобождаване на заделената преди това динамична памет, възстановяване на състояние на програмата и др. Ефектът от заключителните действия е противоположен на ефекта на инициализацията. Естествено е да се даде възможност тези действия да се извършат автоматично при разрушаване на обекта. Това се осъществява чрез **деструкторите** на класовете. Деструкторът е член-функция, която се извиква при разрушаване на динамичен обект с оператора delete или при излизане от блока, в който е бил създаден обекта на класа. Един клас може да има явно дефиниран точно един деструктор. Името му съвпада с името на класа, предшествано от символа ‘~’. Типът му е void и той явно не се указва. Освен това, деструкторът не може да има формални параметри. Ако конструкторът или някоя член-функция на един клас реализира динамично заделяне на памет за някоя член-данна, използването на деструктор е задължително, тъй като трябва да се освободи заетата динамична памет.

Създаване на обекти - чрез дефиниция или чрез функциите за динамично управление на паметта

Съществуват два начина за създаване на обекти: чрез дефиниция или чрез функциите за динамично управление на паметта. В първия случай обектът се създава при достигане на неговата дефиниция и се разрушава при излизане от блока, в който е поместена дефиницията (или когато приключи изпълнението на програмата, ако обектът е глобален). Дефиницията, чрез която се създава обект, може да бъде допълнена с инициализация, която може да се реализира чрез извикване на обикновен конструктор или на конструктора за присвояване. Разрушаването на обекта е свързано с извикването на деструктора на класа, ако такъв е явно дефиниран.
Във втория случай създаването и разрушаването на обекта се управлява от програмиста. Създаването става чрез операцията new, а разрушаването чрез оператора delete. Операцията new използва конструкторите на класа, а операторът delete деструктора на класа.
Създаването на масиви от обекти става по два начина.
Първият начин е чрез обикновена дефиниция. При него масивът се създава при достигането на тази дефиниция и се разрушава при излизане от блока, в който е поместена дефиницията (или при приключване на изпълнението на програмата, ако масивът е глобален). При създаването на масива от обекти за всеки елемент на масива се извиква конструкторът по премълчаване, освен ако не е зададена явна инициализация на масива, която представлява списък от обръщения към конструктори на класа. При разрушававането на масива от обекти за всеки елемент на масива се извиква деструктора на класа.{...rat x[10];...}; {rat x[10] = {rat(1, 2), rat(5), 8, rat(1, 7)}}
Вторият начин е чрез функциите за динамично управление на паметта. Отново създаването става чрез new, при това се указва размерът на масива и унищожаването става чрез delete []. При това, при изпълнението на операцията new за всеки обект от новосъздадения масив се извиква конструктора по премълчаване на класа, а при изпълнението на оператора delete[] се извиква деструктора на класа за всеки един от елементите на масива.{rat * px = new rat[10]; delete [] px;} (delete px ще даде грешка, тъй като px е масив от обекти клас в динамичната памет) Както вече споменахме, за масивите, реализирани в динамичната памет не може явно да се задава инициализация.

Оператори. Предефиниране на оператори

Всеки оператор се характеризира с позиция на оператора, спрямо аргументите му, приоритет и асоциативност. Позицията на оператора спрямо аргументите му го определя като префиксен (поставя се пред аргументите), инфиксен (поставя се между аргументите) или постфиксен (поставя се след аргументите). Приоритетът определя реда на изпълнение на операторите в операторен терм. Операторите с по-висок приоритет се изпълняват преди тези с по-нисък. Асоциативността определя реда на изпълнение на оператори с еднакъв приоритет в операторен терм. В C++ има ляво и дясно асоциативни оператори. Лявоасоциативните оператори се изпълняват отляво надясно, а дясноасоциативните се изпълняват отдясно наляво.
В C++ не могат да се дефинират нови оператори, но всеки съществуващ оператор, с изключение на ::, ?:, ., .*, sizeof, може да бъде предефиниран от програмиста, стига поне един операнд на оператора да е обект на някакъв клас. Предефинирането се извършва чрез дефиниране на специален вид функции, наречени **операторни функции**. Последните имат синтаксиса на обикновени функции, но името им се състои от запазената дума operator, последвана от мнемоничното означение на

предефинираната операция. Когато предефинираната операция изисква достъп до компонентите на класове, декларирани като private или protected, операторната дефиниция трябва да е член-функция или функция-приятел на тези класове. В противен случай, т.е. когато операторната дефиниция е външна функция неприятел на тези класове тя няма достъп до въпросните компоненти. Позицията спрямо аргументите, приоритета и асоциативността на една операция не може да се променят при нейното предефиниране. Също така, броят на нейните аргументи не може да бъде променен и операторните функции не могат да имат аргументи по премълчаване.
В различни случаи операторните функции е най-добре да бъдат приятелски функции или функции-елементи.
Ако левият операнд на предефинираната операция трябва задължително да бъде обект на клас или псевдоним на обект на клас, тогава тя се предефинира с функция елемент. Например операциите за извикване на функция '()', за достъп до елемент на масив '[]', указателната операция '->' и операцията за присвояване '=' винаги се предефинират с функция елемент на клас.
Ако левият операнд трябва да бъде обект от друг клас или от вграден тип, тогава операцията не може да бъде предефинирана с функция-елемент и тя се предефинира с външна функция, която е приятел на класа.
Унарна операция може да се предефинира с помощта на нестатична функция елемент без аргументи или с външна функция-приятел с един аргумент. Този аргумент трябва да бъде или обект на класа или псевдоним на обект на класа.
Бинарна операция може да се предефинира с помощта на нестатична функция елемент с един аргумент или с външна функция-приятел с два аргумента. Единият от тези аргументи трябва да бъде или обект от класа или псевдоним на обект от класа.
Операциите = и унарната & могат да се използват с обекти от всеки клас без те да са предефинирани. По премълчаване, унарната операция &, приложена към обект от кой да е клас, връща адреса на обекта в паметта. По премълчаване, присвояване (=) може да се извършва между обекти от един и същ клас и то се свежда до побитово копиране на данните-елементи на класа. Такова копиране е опасно за класове с данни-елементи, които сочат към динамично разпределена памет. След извършване на такова присвояване за два различни обекта от такъв клас, тези два обекта ще сочат към една и съща област от паметта. Изпълнението на деструктора на кой да е от тези обекти ще доведе до освобождаването на тази памет. Ако после, обаче, чрез другия обект се извърши обръщение към сочената от него вече освободена памет, резултатът ще бъде неопределен.
Поради тази причина, операцията = за такива класове задължително се предефинира.

Шаблони на функции и класове (templates)

Възниква необходимостта от средства, които реализират функции и класове, зависещи от параметри, които задават типове данни и при конкретни приложения параметрите да се конкретизират. Такива средства са **шаблоните**. Те позволяват създаването на функции и класове, използващи неопределени типове данни за своите аргументи. Така шаблоните на класове позволяват да бъдат описвани обобщени типове данни. Шаблоните на класове най-често се използват за изграждане на общоцелеви класове-контейнери (стекове, опашки, списъци и др.). Шаблон на функция се дефинира като обикновена функция, но със следните разлики: заглавието на функцията се предшества от запазената дума template, последвана от списъка от формалните параметри на шаблона, заграден в '<', '>' – списък от идентификатори, предшествани от запазената дума class и разделени със запетаи. Формалните параметри на шаблона означават всеки вграден или потребителски тип и могат да се използват при указването на типове на параметрите на функцията, при указването на типа на резултата на функцията или при указването на тип вътре в тялото на функцията (например, при дефинирани на локални променливи). Използването на дефиниран шаблон на функция става чрез обръщение към обобщената функция, която шаблонът дефинира, с параметри от конкретен тип. При такова обръщение, компилаторът генерира шаблонна функция, като замества параметрите на шаблона с типовете на съответните фактически параметри.
Декларацията на шаблон на клас изглежда като традиционно описание на клас със следната разлика: предшества се от заглавие, което започва с ключовата дума template, последвана от списъка на формалните параметри на шаблона, ограден в '<' и '>'. Този списък има аналогичен вид както при шаблоните на функции.
Формалните параметри на шаблона на класа означават типове (вградени или потребителски) и могат да се използват навсякъде в декларацията на класа – като типове за член-данни, като типове на параметри или тип на резултат на член-функции, като типове на локални променливи на член-функции и т.н. Всяка дефиниция на функция-елемент извън декларацията на класа трябва да се предшества от заглавието на шаблона на класа. Дефиницията е стандартна с тази разлика, че винаги когато в нея се използва името на класа (с изключение на случая, когато то е име на конструктор), то трябва да се задава след него списъка от имената на формалните параметри на шаблона, ограден в '<' и '>'.
За разлика от шаблоните на функции, при създаване на обект на шаблонен клас изрично трябва да се зададат конкретните типове, за които ще се създава този обект, като имената на конкретните типове са подредени в списък, ограден с '<' и '>' след името на класа в дефиницията на обекта. Например, нека имаме следната декларация на шаблон на клас:

```
template <class T, class S>
class Stack {
…
};
```

може и с подразбиращ се тип:

```
template<class T, class S = int>
class Stack{
…
};
```

Тогава са валидни следните дефиниции на обекти:

```
Stack <int, double> obj1;
Stack <double, int> obj2; и т.н.
```

В шаблоните на функции и класове има възможност да се използват и нетипови параметри – те се задават заедно с типовите

параметри в списъка на формалните параметри на шаблона.
Фактическите стойности на нетиповите параметри трябва да са константни изрази, тъй като те се обработват по време на компилация.

Обектно-ориентирано програмиране (C++): Наследяване и полиморфизъм.

Производните класове и наследяването са една от най-важните характеристики на обектно-ориентираното програмиране. Като се използва механизмът на наследяването от съществуващ клас може да се създаде нов клас. Класът, от който се създава, се нарича **базов клас**, класът, който се създава се нарича **производен клас**. Производният клас може да наследи компонентите на един или на няколко базови класа. В първия случай наследяването се нарича **просто**, във втория случай се нарича **множествено**. Дефинирането на производни класове е еквивалентно на конструирането на йерархии от класове. Всеки производен клас може от своя страна да е базов при създаване на нови производни класове. Ако множество от класове имат общи данни и методи, тези общи части могат да се обособят като базови класове, а всяка от останалите части да се дефинира като производен клас на тези базови класове. Така се прави икономия на памет, тъй като се избягва многократното описание на едни и същи програмни фрагменти.
Производните класове се дефинират като обикновени класове с единствената разлика, че след името на производния клас в заглавието се поставя символът ‘:’, последван от списък от двойки [<атрибут_за_област>] <име_на_базов_клас>, разделени със запетаи. Този списък определя кои са базовите класове. Атрибутът за област е незадължителен и може да бъде public, private или protected. Той определя областта на наследените компоненти в производния клас. Ако атрибутът за област е пропуснат, подразбира се private. Множеството от компонентите на производния клас се състои от компонентите на всички негови базови класове, заедно със собствените компоненти на производния клас.
Атрибутът за област на базов клас в декларацията на производния клас управлява механизма на наследяване и определя какъв да бъде режимът на достъп до наследените членове.
При атрибут за област public, елементите от тип public на базовия клас се наследяват като елементи от тип public на производния клас и елементите от тип protected на базовия клас се наследяват като елементи от тип protected на производния клас.
При атрибут за област protected, елементите от тип public и protected на базовия клас се наследяват като елементи от тип protected на производния клас.
При атрибут за област private, елементите от тип public и protected на базовия клас се наследяват като елементи от тип private на производния клас.
И при трите вида наследявания, производният клас няма достъп до елементите от тип private на базовия клас.
Външна функция, която не е приятел на производния клас чрез обект от производния клас има достъп само до компонентите от тип public на производния клас – това означава, че такава функция чрез обект на производния клас може да има достъп до наследени компоненти от базовия клас само ако те са от тип public в базовия клас и атрибутът за област е public.
Да отбележим, че член-функциите на базов клас нямат достъп до каквито и да е компоненти на производния клас. Причината е, че когато базовият клас се дефинира, не е ясно какви производни класове ще произхождат от него.
Функциите приятели на производния клас имат същите права за достъп както член-функциите на производния клас – това са достъп до всички собствени компоненти на производния клас и достъп до наследените компоненти от тип public и protected на базовия клас. Декларацията за приятелство не се наследява – функция приятел на базов клас не е автоматично приятел на производния клас (освен ако не е изрично декларирана като такава в производния клас).
Базовият и производният клас могат да притежават компоненти с еднакви имена. В този случай, производният клас ще притежава компоненти с еднакви имена. Обръщението към такава компонента чрез обект от производния клас или от член-функция на производния клас извиква декларираната в производния клас компонента, т.е. името на собствената компонента скрива името на наследената компонента. За да се използва покритата компонента, трябва да се укаже нейното пълно име: <име_на_клас>::<име_на_компонента>, където <име_на_клас> е името на базовия клас.

Наследяване на конструктори
По-нататък под обикновен конструктор ще разбираме конструктор, различен от конструктора за присвояване. Обикновените конструктори, конструкторът за присвояване, операторната функция за присвояване и деструкторът са методи за които не важат общите правила за наследяване. Тези методи (с някои малки изключения) не се наследяват от производния клас. Причината е, че ако се наследят, те биха се грижили само за наследените компоненти на производния клас, но не и за неговите собствени компоненти.
Основен въпрос е как да се реализира инициализирането на наследената част на производния клас. Най-естествено е това да се направи от конструктора на производния клас, но от друга страна този конструктор няма достъп до private компонентите на базовия клас. Затова конструкторите на производния клас инициализират само собствената част на този клас, а наследената част се инициализира от конструктор на базовия клас – това се осъществява като в дефиницията на конструктора на производния клас се укаже обръщение към съответен конструктор на базовия клас. Основен момент е, че обръщенията към конструктор на базови класове се осъществява посредством инициализиращ списък. Този списък се записва в дефиницията на конструктора на производния клас след името на този конструктор и знак ‘:’. Списъкът се състои от обръщения към конструктори на базовите класове, разделени със запетаи. При просто наследяване в този списък присъства най-много едно обръщение към конструктор на единствения базов клас. При множествено наследяване в списъка са указани няколко обръщения към конструктори на базови класове, най-много по едно обръщение към конструктор за даден базов клас. Имената на параметрите на конструктора на производния клас могат да се използват като фактически

параметри в обръщенията към конструкторите на базовите класове в инициализиращия списък.
Дефинирането на обект от производен клас предизвиква създаване на неявни обекти от базовите класове и добавяне към тях на декларираните в производния клас компоненти. Това означава, че първо се извикват конструкторите на базовите класове, а след това се извиква конструкторът на производния клас. Редът, в който се извикват конструкторите на базовите класове, е редът, в който тези класове са посочени в заглавието на производния клас. На този ред не влияе последователността, в която са посочени конструкторите на базовите класове в инициализиращия списък в дефиницията на конструктора на производния клас. Ако производният клас има член-данни, които са обекти, техните конструктори се извикват след изпълнението на конструкторите на базовите класове и преди изпълнението на тялото на конструктора на производния клас. Редът, в който се изпълняват конструкторите на обектите-елементи е редът, в който те са декларирани в тялото на производния клас.
Ако за базовия клас не е дефиниран конструктор, то за него се създава служебен конструктор по премълчаване, но независимо от това наследената част на производния клас остава неинициализирана. В този случай, в инициализиращия списък в дефиницията на конструктора на производния клас не се извършва обръщение към конструктора на този базов клас.
Ако за базовия клас е дефиниран поне един конструктор с параметри и за този клас не е дефиниран конструктор по премълчаване, то за производния клас задължително трябва да е дефиниран конструктор, който в своя инициализиращ списък извършва обръщение към един от дефинираните конструктори на въпросния базов клас. Ако това не е изпълнено, компилаторът съобщава за грешка.
Последният случай е когато за базовия клас е дефиниран поне един конструктор, включително конструктор по премълчаване. Тогава, ако в производния клас е дефиниран конструктор, то в неговия инициализиращ списък може да присъства обръщение към конструктора на базовия клас или да не присъства. Във втория случай се извиква конструкторът по премълчаване на базовия клас. Ако в производния клас не е дефиниран конструктор, тогава за него се създава служебен конструктор по премълчаване, който от своя страна извиква конструкторът по премълчаване на базовия клас. В последния случай, собствените членове на производния клас остават неинициализирани.

Наследяване на деструктори
Деструкторите на един производен клас и на неговите базови класове се изпълняват в ред, обратен на реда на изпълнение на съответните конструктори. Най-напред се изпълнява деструкторът на производния клас, след това деструкторите на обектите-елементи на производния клас и най-накрая деструкторите на базовите класове.
С някои малки изключения, производният клас не наследява от базовия клас конструктора за присвояване и операторната функция за присвояване.

Наследяване на конструктори за присвояване
При конструкторите за присвояване се спазва същият принцип както при обикновените конструктори на производния и базовия клас. Конструкторът за присвояване на производния клас инициализира собствените член-данни на производния клас, а конструкторът за присвояване на базовия клас инициализира наследените член-данни.
Нека в производния клас не е дефиниран конструктор за присвояване, но в базовия клас има такъв. Тогава, за производния клас се създава служебен конструктор за присвояване, който от своя страна извиква конструктора за присвояване на базовия клас. Да отбележим, че при обикновените конструктори този случай ще предизвика грешка, ако за базовия клас не е дефиниран конструктор по премълчаване. Затова в случая се казва, че производният клас наследява конструкторът за присвояване от базовия клас.
Нека в производния клас и в базовия клас няма дефинирани конструктори за присвояване. Тогава и за двата класа се създават служебни конструктори за присвояване, като този на производния клас извиква този на базовия клас.
Нека в производния клас е дефиниран конструктор за присвояване. В неговия инициализиращ списък може да има, но може и да няма обръщение към конструктор (обикновен или за присвояване) на базовия клас. Препоръчва се в инициализиращия списък на конструктора за присвояване на производния клас да има обръщение към конструктора за присвояване на базовия клас, ако такъв е дефиниран. При това, фактическият параметър на това обръщение може да съвпада с фактическия параметър на обръщението към конструктора за присвояване на производния клас. Това е позволено, тъй като в случая обекта на производния клас може да се разглежда като обект на базовия клас (при работа с обекти във външни функции, това може да се счита само при наследяване от тип public). Ако в инициализиращия списък на конструктора за присвояване на производния клас не е указано обръщение към конструктор на базовия клас, то се извиква конструкторът по премълчаване на базовия клас. Ако базовия клас няма такъв конструктор, компилаторът ще съобщи за грешка.

Наследяване на операторна функция за присвояване
Операторната функция за присвояване на производен клас трябва да указва как да става присвояването както на собствените, така и на наследените си член-данни. За разлика от конструкторите на производни класове, тя прави това в своето тяло (не поддържа инициализиращ списък).
Нека в производния клас не е дефинирана операторна функция за присвояване. Тогава компилаторът създава служебна такава. Тя се обръща към операторната функция за присвояване на базовия клас, чрез която инициализира наследената част, след това инициализира чрез обикновено присвояване и собствената част на производния клас. Затова в този случай се казва, че операторът за присвояване на базовия клас се наследява, т.е. за наследените член-данни се използва служебният или предефинираният оператор за присвояване на базовия клас.
Нека в производния клас е дефинирана операторна функция за присвояване. Тогава тази функция трябва да се погрижи за

присвояването на наследените член-данни. Налага се в нейното тяло да има обръщение към операторната функция за присвояване на базовия клас, ако има такава. Ако това обръщение не е направено явно, то стандартът на езика не уточнява как ще стане присвояването на наследените компоненти.

Множествено наследяване

В случаите когато производният клас наследява повече от един базов клас се казва, че класът е с множествено наследяване. Този вид наследяване е мощен инструмент на обектно-ориентираното програмиране, тъй като чрез него се изграждат графовидни йерархични структури. Имената на базовите класове се задават в заглавието на производния клас след името му и символът ':', разделят се със запетаи, при това всеки от тях се предшества или не от съответен атрибут за област. За член-функциите от голямата четворка на производен клас с множествено наследяване са в сила същите правила, както при производен клас с просто наследяване – тези правила се прилагат независимо към всеки един базов клас. Изпълнението на конструктор на производен клас с множествено наследяване става така: първо се извикват конструкторите на всички базови класове, в реда, указан в заглавието на производния клас – кой конструктор ще се извика зависи от това дали присъства или не обръщение в инициализиращия списък на конструктора на производния клас, второ се извикват конструкторите на собствените обекти-елементи на производния клас, в реда, в който те са описани в тялото на класа и най-накрая се изпълнява тялото на конструктора на производния клас.Извикването на деструкторите става в ред, обратен на реда на съответните конструктори. Препоръчва се конструкторът за присвояване (ако е дефиниран) на производния клас да извършва обръщения чрез инициализиращия си списък към конструкторите за присвояване на всички базови класове.
Операторната функция за присвояване на производен клас с множествено наследяване обикновено има следния вид:

```
<производен_клас>& <производен_клас>::operator= (const <производен_клас>&r) {
        if (this != &r) {
                <базов_клас_1>::operator= (r);
                <базов_клас_2>::operator= (r);
                …
                <базов_клас_N>::operator= (r);
                Del();
                Copy (r);
          }
         return *this;
        }
```

Примерно:

```
A & A::operator= (const A & a){
        if(this != &a){
                delete x;
                x = new  char[strlen(a.x)+1];
                strcpy(x, a.x);
        }
        return *this;
}
```

```
B & B::operator= (const B& b){
        if(this != &b){
        A::operator=(b);
        delete x;
        x = new char[strlen(b.x)+1]; strcpy(x, b.x);
        }
        return *this;
}
```

Тук функцията Del изтрива собствените компоненти на подразбиращия се обект, а функцията Copy (r) копира в подразбиращия се обект компонентите на обекта r, съответни на собствените член-данни на производния клас. Извършва се проверка за самоприсвояване и функцията връща псевдоним на обекта от лявата страна на присвояването, което позволява слепване на няколко присвоявания.

Виртуални базови класове

Виртуалните базови класове са механизъм за отмяна на стандартния наследствен механизъм. При реализиране на йерархии от класове с множествено наследяване е възможно един производен клас да наследи повече от един път даден базов клас. Например, възможно е A да е базов клас за B и C, които от своя страна са базови класове за класа D. Като производни на класа A, класовете B и C наследяват неговите компоненти. От друга страна, класовете B и C са базови за класа D, следователно класът D ще наследи два пъти компонентите на A – веднъж чрез класа B и веднъж чрез класа C. За член-функциите двойното наследяване не е от значение, тъй като за всяка член-функция се съхранява само едно копие. Член-данните, обаче, се дублират и обект на D ще наследи двукратно всяка член-данна, дефинирана в класа A. Многократното наследяване на клас води от една страна до затруднен достъп до многократно наследените членове, а от друга страна до поддържане на множество копия на член-данните на многократно наследения клас, което е неефективно. Преодоляването на тези недостатъци се осъществява чрез използването на виртуални базови класове. Чрез тях се дава възможност да се поделят базови класове. Когато един клас е виртуален, независимо от многократното му участие като базов клас (пряк или косвен), се създава само едно негово копие.
В нашия пример, ако класът A се определи като виртуален за класовете B и C, класът D ще съдържа само един поделен базов клас A. Декларацията на базов клас като виртуален се осъществява като в декларацията на производния клас заедно с името и атрибута за област на базовия клас се укаже и запазената дума virtual. Виртуалното наследяване на един клас указва влияние само на наследниците на неговите непосредствени производни класове. То не указва влияние на тези непосредствени наследници. Дефинирането и използването на виртуални класове има редица особености. Една от тях касае

дефинирането и използването на конструкторите на наследените класове. Нека A е виртуален базов клас за класа B, класът B е базов за класа D, който пък от своя страна е базов за класа E. Ако класът A има конструктор с параметри и няма конструктор по премълчаване, то този конструктор трябва да се извика в инициализиращия списък не само на класа B, но и на конструкторите D и E. С други думи, конструкторите с параметри на виртуални класове трябва да се извикват от конструкторите на всички класове, които са техни наследници, а не само от конструкторите на непосредствените им наследници. Друга особеност е промяната на реда на инициализиране. Инициализирането на виртуалните базови класове предхожда инициализирането на невиртуалните базови класове. Има и още едно уточнение. Нека класът A е базов виртуален клас за класовете B и C, които от своя страна са базови за класа D. Възможно е атрибутът за област на класа A да е public за класа B и private за класа C. Очевидно ситуацията е неоднозначна – външна функция чрез обект на D няма достъп до компонентите на A по пътя A-C-D, но има достъп по пътя A-B-D. Тази неоднозначност се преодолява с избора, че ако в някоя декларация виртуалният клас е наследен като public, счита се, че този клас се наследява като public навсякъде.

Предефиниране на функции – статично и динамично свързване

Възможно е да се използват функции с еднакви имена, в това число и методи на класове – така наречението **предефиниране** на функции. В случая на обикновени функции и собствени член-функции на един и същ клас при извикване на предефинирана функция, кой от нейните варианти се извиква се определя по следния механизъм: по време на компилация се сравняват формалните с фактическите параметри в обръщението и по правилото за най-доброто съвпадане се избира необходимата функция. При член-функциите на йерархия от класове, конфликтът между имената на наследените и собствените методи се разрешава също по време на компилация чрез правилото на локалния приоритет или чрез явно посочване на класа, към който принадлежи метода. В тези два случая тъй като процесът на определяне на реализиране на обръщението към функцията приключва по време на компилация и не може да бъде променян по време на изпълнение на програмата се казва, че има **статично разрешаване на връзката** или **статично свързване**. Ще разгледаме пример, с дървовидна йерархия от три класа – клас Point2 за точка в равнината, клас Point3 за точка в пространството, който наследява пряко Point2 и клас ColPoint3 за оцветена точка в пространството, който наследява пряко Point3.

```
#include <iostream.h>
class Point2 {
 public:
        Point2 (int a = 0, int b = 0){ x = a; y = b;}
        void Print() const { cout << x << “, “ << y; }
 private:
        int x, y;
};
class Point3 : public Point2 {
 public:
        Point3 (int a = 0, int b = 0, int c = 0) : Point2 (a, b) { z = c; }
        void Print () const {
                Point2::Print();
                cout << “, “ << z;
        }
 private:
        int z;
};
class ColPoint3 : public Point3 {
 public:
        ColPoint3 (int a = 0, int b = 0, int c = 0, int o = 0) : Point3 (a, b, c){ col = o; }
        void Print() const {
                Point3::Print();
                cout << “colour: “ << col;
         }
 private:
        int col;
};
```

Сега ще дефинираме примерна функция main.

```
void main () {
        Point2 p2(5, 10);
        Point3 p3(2, 4, 6);
        ColPoint3 p4 (12, 24, 36, 11);
        Point2 *ptr1 = &p3;
        ptr1 -> Print(); cout << endl;
        Point2 *ptr2 = &p4;
        ptr2 -> Print(); cout << endl;
}
```

Резултатът от изпълнението на тази функция е:

2 4
12 24

Изрично ще отбележим, че инициализирането на указателите, които са от типа на базовия клас чрез адреси на обекти от производните класове е възможно, тъй като наследяването и при двата производни класа е от тип public.
И в трите класа е дефинирана функция Print без параметри от тип void. Обръщението ptr1 -> Print(); извежда първите две координати на точката p3, а обръщението ptr2 -> Print(); извежда първите две координати на точката p4, т.е. изпълнява се метода Print() на класа Point2 и в двата случая. Още по време на компилация, член-функцията Print() на Point2 е определена като функция на двете обръщения. Определянето става от типа на двата указателя – той е Point2. Връзката е определена статично и не може да се промени по време на изпълнение на програмата.
Ако искаме след свързването на ptr1 с адреса на p3 да се изпълни член-функцията Print() на Point3, а също след свързването на ptr2 с адреса на p4 да се изпълни член-функцията Print() на ColPoint3, са небходими явни преобразувания от следния вид:

```
((Point3 *)ptr1) -> Print();
((ColPoint3 *)ptr2) -> Print();
```

В този случай връзките отново са разрешени статично.
При статичното свързване по време на създаването на класа трябва да се предвидят възможните обекти, чрез които ще се викат неговите член-функции. При сложни йерархии от класове това е не само трудно, но понякога и невъзможно.
Динамично свързване и виртуални функции
Езикът C++ поддържа още един механизъм, прилаган върху специален вид член-функции, наречен **динамично свързване.** При него изборът на функцията, която трябва да се изпълни, става по време на изпълнение на програмата. Динамичното свързване капсулира детайлите в реализацията на йерархията. При него не се налага проверка на типа. Разширяването на йерархията не създава проблеми. Това обаче е с цената на забавяне на процеса на изпълнение на програмата. Прилагането на механизма на динамичното свързване се осъществява върху специални член-функции на класове, наречени **виртуални член-функции** или само **виртуални функции**. Виртуалните методи се декларират чрез поставяне на запазената дума virtual пред декларацията им, т.е. virtual <тип_на_резултата> <име_на_метод> (параметри);
Да предположим, че е в горния пример член-функцията Print() и в трите класа е декларирана като виртуална. Тогава при обръщенията ptr1 -> Print(); и ptr2 -> Print(); коя функция ще бъде извикана се определя по време на изпълнение на програмата. Определянето е в зависимост от типа на обекта, към който сочи указателя, а не от класа към който е указателя. В случая, указателят ptr1 е към класа Point2, но сочи обекта p3, който е от класа Point3 и затова обръщението ptr1 -> Print(); води до изпълнение на функцита Print() от класа Point3. Аналогично, указателят ptr2 е към класа Point2, но сочи обекта p4, който е от класа ColPoint3 и затова обръщението ptr2 -> Print(); води до изпълнение на функцията Print() от класа ColPoint3.
Ще отбележим някои важни особености на виртуалните функции:

1. Само член-функциите на класове могат да се декларират като виртуални. По технически причини конструкторите не могат да се дефинират като виртуални, но е възможно да има виртуални деструктори.
2. Ако в даден клас е декларирана виртуална функция, декларираните член-функции със същия прототип (име, параметри и тип) в производните на класа класове автоматично стават виртуални, дори ако запазената дума virtual не присъства в тяхните декларации.
3. Ако в производен клас е дефинирана функция със същото име като определена вече в базов клас като виртуална член-функция, но с други параметри или тип, това ще бъде друга функция, която може да бъде или да не бъде декларирана като виртуална.
4. Възможно е виртуална функция да се дефинира извън клас. Запазената дума virtual обаче присъства само в декларацията на функцията в тялото на класа, но не и в нейната дефиниция.
5. Виртуалните функции се наследяват като останалите компоненти на класовете.
6. Виртуалната функция, която в действителност се изпълнява зависи от типа на аргумента.
7. Виртуалните функции не могат да бъдат декларирани като приятели на други класове.

Всяка член-функция на клас, в който е дефинирана виртуална функция има пряк достъп до виртуалната функция, т.е. на локално ниво достъпът се определя по традиционните правила.
На глобално ниво достъпът се определя от вида на секцията, в която е дефинирана виртуалната функция на класа към който сочи указателят, чрез който се активира функцията, а не от вида на секцията, в която е предефинирана виртуалната функция на класа, към който принадлежи обекта, сочен от въпросния указател.
Съществуват три случая при които обръщението към виртуална функция се разрешава статично (по време на компилация):

1. Виртуалната функция се извиква чрез обект на класа, в който е дефинирана.
2. Виртуалната функция се извиква чрез указател към обект, но явно, чрез бинарната операция за разрешаване на достъп :: е посочена конкретната функция.
3. Виртуалната функция се активира с тялото на конструктор или деструктор на базов клас.

Виртуални функции -> Полиморфизъм

Основно предимство на виртуалните функции е, че чрез тях могат да се реализират полиморфни действия. **Полиморфизмът** е важна характеристика на обектно-ориентираното програмиране и тя се изразява в това, че едни и същи действия се реализират по различни начини в зависимост от обектите, към които се прилагат, т.е. действията са полиморфни (с много форми). Полиморфизмът е свойство на член-функциите на обектите и в C++ се реализира чрез виртуални функции. За да се

реализира полиморфно действие, класовете върху които то ще се прилага трябва да имат общ родител или прародител, т.е. да бъдат производни на един и същи базов клас. В този клас трябва да бъде дефиниран виртуален метод, съответстващ на полиморфното действие. Във всеки от производните класове този метод може да бъде предефиниран съобразно особеностите на този клас. Активирането на полиморфното действие става чрез указател към базовия клас, на който могат да се присвоят адресите на обекти на който и да е от производните класове от йерархията. Ще бъде изпълнен методът на съответния обект, т.е. в зависимост от обекта, към който сочи указателят ще бъде изпълняван един или друг метод. Ако класовете, в които трябва да се дефинират виртуални методи нямат общ родител, такъв може да бъде създаден изкуствено чрез дефиниране на така наречен абстрактен клас.

Абстрактен клас

Възможно е виртуалните функции да имат само декларация без да имат дефиниция. Такива виртуални член-функции се наричат **чисти**. За да се определи една виртуална функция като чиста се използва следния синтаксис:
virtual <тип><име_на_функция>(<параметри>) = 0;
Клас, който съдържа поне една чисто виртуална функция се нарича **абстрактен клас**. Основното свойство на абстрактните класове е, че не е възможно от тях да се създават обекти. Чистите виртуални функции могат да се предефинират в производните класове със същите прототипи, а може и да не се предефинират. Когато производния клас наследи чисто виртуална функция без да я предефинира, той също става абстрактен. В случай, че производния клас предефинира всички чисто виртуални функции на своя базов клас, той става **конкретен** и от него могат да се създават обекти.
Абстрактните класове са предназначени да служат като базови на други класове. Чрез тях се обединяват в обща структура различни йерархии. Обикновено, абстрактните базови класове определят интерфейса за различни типове обекти в йерархията на класовете. Всички обработки в йерархията могат да прилагат един и същ 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.

Приятелски класове и функции

Често е необходимо съвместното използване на два класа. Обектно-ориентираното програмиране налага капсулиране на данните. Достъпът до private компонентите на даден клас от функция извън класа е забранено. Това може да се окаже затруднение. Например ако искаме да реализираме функция, която пресмята произведението на матрица с вектор, ще трябва да имаме достъп и до членовете и на двата класа. Един начин е да се направят член-данните и на двата класа public. Това ще доведе до загубване на предимствата на капсулирането. Друг начин е да се използват public функции на достъп, осъществяващи достъп до стойностите на член-данните. Това води до забавяне на изпълнението на програмата. Трети начин е декларирането на функции или класове – приятели на клас. Приятелите на даден клас (функции или класове) имат достъп до всички негови компоненти, т.е. членовете на класа са винаги public за функциите приятели. Ако клас е деклариран като приятел, всички негови член-функции стават функции приятели.