 ХИМИКОТЕХНОЛОГИЧЕН И МЕТАЛУРГИЧЕН УНИВЕРСИТЕТ – СОФИЯ

**ИНФОРМАТИКА   
част трета**

**Обектно-ориентирано програмиране на С++**

**лектор: гл. ас. д-р Стефан М. Панов   
Катедра “Информатика”**

**Лекция 2**

**Подробно за класовете**

Функции- приятели

В C++ съществува възможност да разрешим достъпа към **закритите** членове на един клас **на функции, които не са членове на тоя клас**. За целта такава функция трябва да се обяви като “приятел” по отношение на разглеждания клас. **Това става като нейният прототип се включва в public-раздела на класа, предшестван от ключовата дума** **friend**. Например:

**class cl {**

**// . . .**

**public:**

**friend void frnd(cl ob);**

**// . . .**

**};**

Да разгледаме кратък пример: **prog2\_1.cpp**, дадена отделно, в която функцията-”приятел” **sum()** се използва за достъп до закритите членове на класа **myclass**. Тя има пълен достъп до **private**-членовете на класа **myclass**. **Обърнете внимание и на това, че функцията sum() се вика по обичайния начин, т.е. без привързване към обект (без използване на оператора “точка”).** **Понеже тя не е функция-член, то при нейното извикване не   
трябва да се задава името на обекта**.

Съществуват определени обстоятелства, при които функциите-”приятели” има голямо значение. **Първо**, функциите-”приятели” могат да бъдат полезни **при презареждане на операторите** на определени типове. **Второ**, функциите-”приятели “ **опростяват създаването на някои функции за вход-изход**. За това ще стане дума скоро. **Третата причина** за използване на функции-”приятели” се състои в това, че в някои случаи два или повече класа могат да съдържат членове, които се намират във взаимовръзка с други части на програмата. **Например**: Имаме два различни класа, които при възникване на определени събития показват на екрана “изскачащи” съобщения. Други части на програмата, които са предназначени за извеждане на данните на екран са длъжни да знаят, дали “изскачащото” съобщение е активно, да не би случайно да го презапишат. Във всеки клас може да създадем **функция-член**, връщаща стойност, по която може да се съди затова, активно ли е съобщението или не; но проверката на такова условие изисква допълнителни разходи (т.е. две извиквания на функцията вместо едно). Ако е необходимо статусът на "изскачащо" съобщения да се проверява често, тези допълнителни разходи могат да се окажат просто неприемливи. Обаче с помощта на функция, **"приятел" и на двата класа**, може директно да се проверява статусът на всеки обект. Тая идея е илюстрирана в следващата програма **Prog2\_2.cpp**.

Кратък коментар върху предложената **Prog2\_2.cpp**:

Обърнете внимание на това, че в програмата се използва **изпреварващо обявление** на класа **С2**. Неговата необходимост е обусловена от това, че при обявяването на функция **idle()** в класа **С1** се използва указател към клас **С2** преди неговото (на **С2)** обявление.

**Изпреварващото обявление е предназначено за обявяване на името на типа на клас преди определението на самия клас.**

**“Приятел” на един клас може да бъде член на друг клас.** Ще пренапишем предишната програма така, че функцията **idle()** да стане член на класа **С1**.

Но това ще направим по време на упражнения, тук помагаме с една инструкция от новото решение:

**public: int idle(C2 b); // сега idle е член на класа C1**

**...**

Понеже функцията **idle()** е член на класа **C1**, тя има пряк достъп до променливата **status** на обекти от типа **С1**. Следователно, като параметър към функцията **idle()** е необходимо да се предава само обект от типа С2.

Презареждане на конструктори

Независимо от факта, че конструкторите могат да изпълняват уникални действия, те не се отличават силно от другите типове функции и **също така могат да се презареждат.**

**За да се презареди конструктор на клас е достатъчно той да се обяви във всички нужни формати и да се определи всяко действие, свързано със съответния формат.** Например, в следващата програма **Prog2\_3.cpp** се обявява клас **timer**, който действа като намаляващ таймер. **В тоя пример конструкторът се презарежда три пъти**, предоставяйки по такъв начин възможността да се задава времето както в секунди (при това или с число, или с низ), така и в минути и секунди. В тази програма се използва стандартната библиотечна функция **clock(),** **която връща броя сигнали, приети от системния часовник от момента на начало на изпълнение на програмата.** Ето как изглежда прототипът на тая функция: **clock\_t clock();**

Типът **clock\_t** е разновидност на дългия целочислен тип. Операцията деление на стойността, връщана от функцията **clock (),** на стойността **CLOCKS\_PER\_SEC** позволява да се преобразува резултатът в секунди.

При създаването (във функция **main ())** на обектите **а, b** и **с** от класа **timer** им се присвоява начална стойност по три различни начини, **поддържани от презаредените конструктори.**

**Във всеки от случаите се извиква конструкторът, който съответства на зададения списък от параметри и точно затова той правилно инициализира “своя” обект**.

Динамична инициализация

В **C++** както **локалните, така и глобалните променливи могат да бъдат инициализирани по време на изпълнение на програмата**. **Тоя процес е прието да се нарича динамична инициализация**. Променлива може да се инициализира (по време на изпълнение на програмата), използвайки всеки **С++-**израз, **действителен към момента на обявяването на тази променлива**. Например, следващите варианти на инициализация на променливи са напълно допустими в **C++:**

**int n = strlen(str);**

**double arc = sin(theta);**

**float d = 1.02 \* count / deltax;**

Динамична инициализация при конструктори

Подобно на простите променливи и **обектите могат да се инициализират динамично при тяхното създаване. Това средство позволява да се създава обект от нужния тип с използване на информация**, **която става известна само по време на изпълнение на програмата**. За да покажем, как работи механизмът на динамична инициализация ще модифицираме програмата, реализираща таймера.

Да си припомним, че в първия вариант на програмата ние не получихме голямо преимущество от презареждането на конструктора **timer(),** защото всички обекти от тоя тип се инициализираха с помощта на константи, известни по време на компилация на програмата. Но в случаите, когато обектът е необходимо да се инициализира по време на изпълнение на програмата може да се получи съществена полза от наличието на множество различни формати за инициализация.

**Това позволява на програмиста да избере от съществуващите конструктори тоя, който най-точно съответства на текущия формат на данните**.

Например, в новата версия на **main()** обектите **b** и **с** използват динамична инициализация.

**int main() {**

**timer a(10); a.run();**

**reencode(“Въведете броя секунди: “);**

**char str[80]; cin >> str;**

**timer b(str); // динамична инициализация**

**b.run();**

**reencode(“Въведете минути и секунди: “);**

**int min, sec; cin >> min >> sec;**

**timer c(min, sec); // динамична инициализация**

**c.run (); return 0; }**

Трудно може да не се съгласим с това, че **наличието на множество формати за инициализация избавя програмиста от изпълнение на допълнителни преобразования при инициализацията на обекти**.

**Механизмът за презареждане на конструктори способства за понижение на степента на сложност на програмите, позволявайки създаването на обекти по начин, който най-добре съответства на тяхното използване**.

От друга страна, претрупването на кода с конструктори за обработка на **рядко възникващи ситуации** оказва, като правило, **дестабилизиращо** влияние на програмата.

**Да запомним.** **Разработвайки презаредени конструктори е необходимо да решим какви ситуации е важно да предвидим и какви може да не отчитаме.**

Присвояване на обекти

**Ако два обекта имат еднакъв тип (т.е. те и двата са обекти на един и същи клас), то един обект може да се присвои на друг**. **Ако един обект се присвоява на друг, то по подразбиране данните на първия обект побитово се копират във втория**.

Присвояването на обекти е показано в следващата програма **Prog2\_4.cpp**.

**Да запомним**: **Присвояването един обект на друг прави техните данни идентични, но** **тези два обекта остават напълно независими**.

Както ще бъде показано по-надолу**, операторът за присвояване може да се презареди, дефинирайки собствени операции за присвояване.**

Предаване на обекти към функции

**Обект може да се предава към функция точно така, както стойността на всеки друг тип данни,** а именно чрез предаване на параметри по стойност. **Тоест, на функцията се предава не самият обект, а негово копие.** **Следователно, измененията внесени в обекта при изпълнение на функцията не оказват никакво влияние на обекта, използван като аргумент на функцията**. Тоя механизъм е показан в следващата програма **Prog2\_5.cpp**, където изменението на **локалния** обект **х** не се отразява на обекта **о**.

Конструктори, деструктори и предаване на обекти

Независимо от това, че предаването към функции на несложни обекти в качеството на аргументи е достатъчно проста процедура, **все пак могат да възникнат непредвидени събития, имащи отношение към конструкторите и деструкторите**. За да си изясним това нека да разгледаме следващата програма **Prog2\_6.cpp**. В нея забелязваме две обръщения към деструктора.

**Когато при извикване на функция се създава копие на аргумента, обичайният конструктор не се вика**. **Вместо него се извиква конструкторът на копие на обекта.** **Конструкторът на копие (копиращият конструктор) определя как трябва да бъде създадено копие на обекта. (Как точно ще се покаже надолу.) Но ако в класа явно не е определен конструктор на копие, C++ го предоставя по подразбиране. Копиращият конструктор по подразбиране създава побитово (т.е. идентично) копие на обекта.**

**А защо не се вика обикновения конструктор:** **Понеже обичайният конструктор се използва за инициализации на някои аспекти на обекта**, **той не трябва да се вика за създаването на копие на вече съществуващ обект**. **Такова извикване би изменило неговото съдържание. При предаване на обект към функция има смисъл да се използва текущото състояние на обекта, а не неговото начално състояние**.

**Но когато функцията завършва и копието на обекта, използван като аргумент се разрушава, за целта се извиква деструкторът на тоя обект.** **Необходимостта от извикване на деструктора е свързана с излизане на обекта извън областта на видимост**. Именно затова в предишната програма има две обръщения към деструктора.

Потенциални проблеми при предаване на параметри

Независимо от това, че обектите се предават към функциите “по стойност”, което теоретически защитава аргумента и го изолира от приемащия параметър, тук все-пак е възможен страничен ефект или даже заплаха за “живота” на обекта- **аргумент**.

**Такъв е случаят, когато обектът, използван като аргумент изисква динамично заделяне на памет**. При освобождаване на тая памет при разрушаване, т.е. **при извикване на деструктора, неговото локално копие ще освободи същата област от паметта, която е била заделена за оригиналния обект**. **И тоя факт става голям проблем**, защото оригиналният обект все още използува тази (вече освободена) област от паметта. Описаната ситуация прави изходния обект “повреден” и по същество, непригоден за употреба. Да разгледаме следващата програма **Prog2\_7.cpp**.

**Тая програма съдържа принципиална грешка.** И ето защо: при създаване във функцията **main()** на обекта **а** се заделя област от паметта, адресът на която се присвоява на указателя **а.р**. При предаване към функцията **display()** обектът **а** **се копира** в параметъра **ob**. **Това означава, че и двата обекта (а и ob) ще имат еднакви стойности за указателя р**.

С други думи, в двата обекта (в оригинала и копието му) членът-данна **р** ще указва на една и съща динамично заделена област от паметта. **При завършване на функцията display() обектът ob се разрушава и неговото разрушение се съпровожда от извикването на деструктора. Деструкторът освобождава областта от паметта, адресирана от указателя ob.p.** Но тая (вече освободена) област от паметта е **същата** област, към която все още указва **а.р** от изходния обект! Налице е сериозна грешка.

**В действителност нещата са още по-зле**. При завършване на програмата се разрушава обектът **а** и динамично заделената (още при неговото създаване) памет се освобождава **вторично. Работата е там, че повторното освобождаване на една и съща динамично заделена област от паметта се счита за неопределена операция, която по правило (в зависимости от това, как е реализирана системата за динамично разпределение на паметта), предизвиква неотстранима грешка**.

Може би някои вече се досещат, че един от начините за решение на проблема се състои не в предаване на самия обект, а на указател към него или на псевдоним. **В такъв случай не се създава копие на обекта и деструкторът не се вика**. Т.е. за конкретния случай едно добро решение е да сменим реда

**void display(myclass ob){** със

**void display(myclass &ob){**

Предаването на псевдоним на обекта е прекрасно решение на описания проблем, но не е универсално лекарство за всички проблемни ситуации.   
**За щастие, има и по-общо решение: може да се създаде собствена версия на копиращия конструктор**. Това позволява точно да се определи, как да се създава копието на обекта и с това именно да се избегнат описаните   
по-горе затруднения. Но преди да се заемем с конструктора на копие има смисъл да се разгледа още една ситуация.

Връщане на обекти от функции

След като може да се предават обекти към функции, то “със същия успех” функциите могат да връщат обекти. За да може една функция да върне обект: **първо,** необходимо е да се обяви като тип на връщаната от нея стойност типът на съответния клас. **Второ**, нужно е да осигурим връщане на обекта от тоя тип посредством инструкция **return**. Да разгледаме програма **Prog2\_8.cpp**, тя е пример за функция, която връща обект. В този пример функцията **input()** създава **локален** обект **str** на класа **sample**, а после прочита низ от клавиатурата. Низът се копира в низа **str.s**, след което обектът **str** се връща от функцията **input()** и се присвоява на обекта **ob** във функция **main().**

**Връщането на обекти от функции крие обаче потенциални проблеми в следния сценарий:**

**Ако функция връща класов обект и ако се създава временен обект, който да съхранява връщаната стойност, именно той (обекта) се връща от функцията. След връщане на стойността обектът се разрушава.   
Обаче, разрушаването на временния обект може в някои ситуации да предизвика непредвидени странични ефекти.(не винаги се създава временен обект)**

Такъв **може да е** случаят когато обект, върнат от функция, има деструктор, който освобождава динамично заделената памет. Тая памет ще бъде освободена даже в случай, когато обектът, получаващ върнатата от функцията стойност, все още я използва. Да разгледаме **Prog2\_9.cpp**, която може да е некоректна версия на предишната програма. **Какво точно ще се случи зависи от това, как конкретният компилатор реализира динамичното заделяне на памет.** Но по идея деструкторът трябва да се извика 2 или 3 пъти – един път при края на **input()** за локалния обект **str**, втори път за временния обект (ако такъв се създава, това зависи от компилатора и напоследък не се случва) и трети път за обекта от **main()**. Но и в двата (трите) случая ще бъде направен опит за освобождаването на една и съща памет. **Обаче опит да се освободи вече освободена памет води до грешка (не за всички, защото компилаторите се развиват и за някои от тях като в devC++ към 2021г. грешка не се наблюдава)**. Ако има такъв проблем той се решава или ако вместо самия обект се върне указател към обекта (или псевдоним на обекта) или както вече се спомена по-рано **да се създаде собствена версия на копиращия конструктор.**

Създаване и използване на копиращ конструктор

**Конструкторът на копие** (**копиращият конструктор**) **представлява специален тип презареден конструктор**. **Той** **позволява да се управляват действията, от които се състои процесът на създаване на копие на обекта**.

В сърцевината на разглежданите проблеми лежи създаване на побитово копие на обекта. За да се предотврати тяхното възникване е **необходимо точно да се определи, какво трябва да се прави при създаване на копие на обекта и по този начин да се избегнат нежелателните странични ефекти**.

Преди подробното запознаване с копиращия конструктор е важно да се разбере, че **в C++ са определени два отделни вида ситуации, в които стойността на един обект се предава на друг**.

**Първата такава ситуация е присвояването, а втората — инициализацията**. Инициализация се прави по три начина, т.е. в три случая, когато:

■ един обект явно инициализира друг обект, например, в обявление;

■ копия на обект се предава на параметъра на функция;

■ генерира се временен обект (най-често като стойност, връщана от функция).

**Копиращият конструктор се използва само при инициализации. Той не се ползва при присвояване**.

Ето как изглежда най-разпространеният формат на конструктора на копие.

**име\_на\_класа (const име\_на\_класа &obj) { // тяло на конструктора}**

Тук елементът **&obj** означава алтернативно име (псевдоним) на обекта, който се използва за инициализация на друг обект. Например, при изпълнение на следващите инструкции ще се извика копиращият конструктор на класа **myclass**.

**myclass х = у;** **// Обектът y явно инициализира обекта x**

**х.fun1(у);** **// Обектът у се предава като аргумент**

**у = fun2();** **// Обектът у приема обекта, връщан от функцията fun2.**

В първите два случая към копиращия конструктор ще се предаде **псевдоним** на обекта **у**, а в третия — псевдоним на обекта, връщан от функцията **fun2().**

За да се разбере в дълбочина предназначението на копиращия конструктор, ще се разгледа подробно неговата роля във всяка от тия три ситуации.

Копиращ конструктор и параметри на функция

При предаване на обект към функции като аргумент се създава копие на обекта. **Ако в класа е определен конструктор на копие, то именно той се извиква за създаване на копието.** Да разгледаме програма **Prog2\_10.cpp**, в която се използва копиращ конструктор за правилна обработка на обектите при тяхното предаване към функцията като аргументи. Това е коректната версия на програма **Prog2\_7.cpp**, представена по-рано.

**При изпълнение на програмата става следното**: когато във функция **main()** се създава обект **а**, обикновеният конструктор заделя памет и адресът на тая област от паметта се присвоява на указателя **а.р**. После обектът **а** се предава на функцията **display(),** а именно— на нейния параметър **ob**. **В този случай се извиква копиращият конструктор, който създава копие на обекта а.** Копиращиятконструктор заделя памет за това копие, а стойността на указателя към заделената област от паметта се присвоява на члена **р** на обекта-копие. **После стойността, адресирана от указателя р на изходния обект се записва в областта от паметта, сочена от указателя р на обекта-копие.** **По такъв начин, областите от паметта, адресирани от указателите**   
**а. р** **и** **ob.р**, **са разделени и независими една от друга**, **но съдържащите се в тях стойности** (към които указват **а.р** и **ob.р**) **са еднакви**.

При завършване на функция **display()** обектът **ob** излиза извън областта на видимост. Това излизане се съпровожда с извикване на неговия деструктор, който освобождава областта от паметта, сочена от указателя **ob.p**. Накрая, при завършване на функция **main()** обектът **а** излиза извън областта на видимост, което също се съпровожда с извикване на неговия деструктор и съответното освобождаване на областта от паметта, сочена от указателя **а.р**. Както се вижда, използването на копиращ конструктор отстранява деструктивните странични ефекти, свързани с предаване на обект към функция.

Копиращ конструктор при инициализация на обекти

**Копиращият конструктор се извиква и в случаите, когато един обект се използва за инициализация на друг**.

Да разгледаме следващата програма **Prog2\_11.cpp**. При изпълнение на програмата се вижда, че когато обектът **а** се използва за инициализация на обекта **b** се извиква копиращият конструктор. Използването на копиращ конструктор гарантира, че **обектът b ще задели за своите членове данни собствена област от паметта.** Без копиращия конструктор обектът **b** просто би представлявал точно копие на обекта **а, а членът а.р би указвал към същата област от паметта, както и члена b.р**.

**Да запомним, че при присвояване се вика обикновеният конструктор**, както става в следващият код:

**myclass а (2), b(3); ...; b = а;**

Използване на копиращ конструктор при връщане на обект

**Копиращият конструктор също се извиква при създаване на временен обект, който се явява резултат на връщането на обект от функция.**   
Да разгледаме следващата програма **Prog2\_12.cpp**.**!! трябва да се извика конструкторът на копието, но в devC++ и Code::Blocks той не се вика! Във VS 2013 се вика !**

Още за копиращите конструктори

C++ е много мощен език, но е сложен. Копиращите конструктори са механизмът, който много програмисти дават като основен пример за сложността на езика, доколкото това средство не се възприема на интуитивно равнище. За мнозина не става веднага очевиден отговорът на въпроса: кога е нужен копиращ конструктор и кога не. Тая ситуация често се изразява в такава форма: “А не съществува ли по-просто начин?”. Отговорът също не е прост: **и да, и не!** **Езици като Java и С#, нямат копиращи конструктори, защото и в двата не се създават побитови копия на обекти. Работата е в това, че както Java, така и C# динамично заделят памет за всички обекти, а програмистът оперира с тия обекти изключително чрез псевдоними**. **Ето защо при предаване на обекти като параметри на функции или при връщането им от функции няма необходимост от копия на обекти**. Този факт прави тия езици по-прости, но за тая простота също трябва да се плати. **Работата с обекти изключително посредством псевдоними (а не пряко, както е в C++) налага ограничения на типовете операции, които може да изпълнява програмистът**. Нещо повече, такава употреба на обектни псевдоними в **Java** и **C#** не позволява точно да се определи, кога обектът ще бъде разрушен. **В C++ обектът винаги се разрушава при излизане извън областта на видимост.**

**Езикът C++ предоставя на програмиста пълен контрол над ситуациите, възникващи в програмата.** Точно затова той е по-сложен в сравнение с **Java** и **С#**. Това е цената, която плащаме за мощността на програмиране.

Ключовата дума this

**Ключовата дума this е указател към обекта извикващ функция-член.**

**При всяко извикване на функция-член към нея автоматично се предава указател, именуван с ключовата дума this, сочещ към обекта, за който се вика тая функция.**

**Указателят this е неявен параметър за всички функции-членове**.

Както знаем, една функция-член има пряк достъп до закритите (**private**) членове данни на своя клас. Например, ако в дадена функции-член имаме:

**i = 10;** в действителност горната инструкция представлява съкратена форма на **this->i = 10;** Което е показано и в програма **Prog2\_13.cpp**.

**Важно! Функциите-”приятели” нямат указател this, понеже те не са членове на класа**. Край на лекция 2.