 ХИМИКОТЕХНОЛОГИЧЕН И МЕТАЛУРГИЧЕН УНИВЕРСИТЕТ – СОФИЯ

**ИНФОРМАТИКА   
част трета**

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

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

**Лекция 1**

**Въведение в класовете**

Основни понятия за класа

Класът е фундаментът, върху който е построена **С++-**поддръжката на **обектно-ориентираното програмиране**, а също и ядрото на много по- сложни програмни средства. Класът е базовата единица за инкапсулация, обезпечаваща механизма за създаване на обекти.

**Класът определя нов тип данни, който задава формата на обекта. Един клас включва както данни, така и код, предназначен за изпълнение върху тези данни. Следователно, класът свърза данните с кода**. **В C++** **спецификацията на класа се използва за построяването на обекти.** **Обектите — това са екземпляри на класа**. По същество, класът представлява набор от планове, които определят как се строи обекта**. Важно е да се разбира, че класът е логическа абстракция**, **която реално не съществува дотогава, докато не бъде създаден обект от тоя клас**, т.е. това, което ще стане физическо представяне на същия клас в паметта на компютъра.

Данните в класа се обявяват във вид на променливи, а кодът се оформя във вид на функции**. Функциите и променливите, съставящи класа, се наричат негови членове.**

**Клас се създава с помощта на ключовата дума class.** Обявяването на клас е синтактично подобно на обявяване на структура. Да разгледаме пример. Следващият клас определя типа **queue**, предназначен за реализация на опашка (FIFO).

class queue { // Създаване на класа queue  
    **int** q[100];   
    **int** sloc, rloc;   
    **public**: void **init** ();   
    void **qput**(**int** i); **int** **qget**(); // също са public !!  
};

Всеки клас може да съдържа както закрити, така и открити членове.

**По подразбиране всички елементи, определени в класа са закрити** **(private-членове).** Например, променливите **q, sloc** и **rloc** са закрити. Това означава, **че достъп до тях могат да получат само други членове на класа queue**; никакви други части на програмата не могат да направят това. В това се състои едно из проявленията на капсулацията: **програмистът в пълна степен да може да управлява достъпа до определени елементи-данни.** Закрити могат да се обявяват и функции (в примера такива няма) - тогава те могат да бъдат извиквани само от други членове на същия клас.

**Ключовата дума public се използва за обявяване на открити членове на класа**. **Всички променливи или функции, определени след спецификатора** **public** **са** **достъпни за всички други функции на програмата**. **Ключовата дума private се използва за обявяване на закритите членове на класа. Всяка функция-член на класа има достъп до закритите елементи на същия клас**.

**За да добавим функция-член в даден клас, трябва да определим нейния прототип в обявлението на тоя клас**.

Определили веднъж класа, можем да създадем обект от този “класов” тип, използвайки името на класа. По такъв начин **името на класа става спецификатор на новия тип.** Например, при изпълнение на следващата инструкция се създават два обекта **Q1** и **Q2** от типа **queue**:

**queue Q1, Q2;**

След създаването на обект на класа той има собствено копие на членовете данни, които съставляват класа. Това означава, че всеки от обектите **Q1** и **Q2** ще има собствени отделни копия на променливите **q, sloe** и **rloc**. Следователно данните, принадлежащи на обекта **Q1**, **са отделени** (изолирани) от данните, принадлежащи на обекта **Q2**.

**За да получим достъп до открит член на класа чрез обект на тоя клас, трябва да използваме оператор “точка”** (също както при работа със структури). Например, така ще изведем на екран стойността на променливата **sloc**, принадлежаща на обекта **Q1** : **cout << Q1.sloc**;

**За да реализираме (дефинираме) функция, която е член на даден клас е необходимо да съобщим на компилатора към кой клас тя принадлежи.** Например, ето как може да запишем кода на функцията **qput ()**.

void queue::**qput**(**int** i){  
 **if** (sloc==100) {  
 cout << "Опашката е пълна.\n"; return;  
 }  
 sloc++; q[sloc] = i;  
}

**Функциите-членове може да извикваме само относно зададения обект**. За да извикаме функция-член от такава част на програмата, която се намира извън класа **е необходимо да използваме името на обекта** **и оператор “точка”.** Например, така се извиква функция **init()** за обекта **ob1**:   
**queue ob1, ob2; ob1.init();**

като действията, определени във функции **init(),** **ще засегнат копието на данните, отнасящи се към обекта ob1**.

**Функция-член** може **непосредствено** да се обръща към **всеки член-данни на своя клас**, но **код**, разположен **извън класа** е **длъжен** да се обръща към променлива на класа, **използвайки име на обекта и оператора “точка”.**

В приведената по-долу програма **Prog1\_1.cpp** класът **queue** е реализиран напълно.

#include <iostream>   
**using** **namespace** std;  
class queue { // Създаване на класа queue  
   **int** q[100] ; **int** sloc, rloc; **public**:  
   void **init** (); void **qput**(**int** i); **int** **qget**();  
};// забележете точката след дефиницията на класа  
 void queue::**init**(){// Инициализация на класа queue  
  rloc = sloc = 0;}  
void queue::**qput**(**int** i){// запис в опашката на целочислена стойност,   
    **if**(sloc==100) {  
        cout << "Опашката е запълнена.\n"; return;  
    }  
    sloc++; q[sloc] = i;  
}  
**int** queue::**qget**() {// Извличане от опашката на целочислена стойност,   
    **if** (rloc == sloc) {  
        cout << " Опашката е празна.\n"; return 0;  
    }  
    rloc++;return q[rloc];  
}  
**int** **main**(){  
    queue a, b; // Създаване на два обекта на класа queue.  
    a.**init**(); b.**init**() ;  
    a.**qput**(10); b.**qput**(19);  
    a.**qput**(20);b.**qput**(1);  
    cout << "Съдържание на опашка a: ";   
 cout<<a.**qget**()<<" ";cout<<a.**qget**()<<"\n";  
    cout << " Съдържание на опашка b: ";

cout<<b.**qget**()<<" ";cout<<b.**qget**()<<"\n" ;  
    return 0;  
}

При изпълнение на програмата ще видим:

Съдържание на опашка a: 10 20

Съдържание на опашка b: 19 1

Не забравяйте, че **закритите членове на класа са достъпни само от функциите, които са членове на същия клас.** Например, инструкцията

**a.rloc = 0;** **не можем** да включим във функция **main()** на нашата програма.

**Общ формат на обявлението на клас**

Общият формат на обявлението на класа има следния вид:  
class **име\_на\_класа** {  
 **private:** // private: може да се пропусне, защо?

Закрити данни и функции

**public**:  
 открити данни и функции

 } **списък\_от\_обекти;**

**но не е задължително** веднага да създаваме обекти, както е в примера.

Достъп до членовете на един клас

Да разгледаме следния прост клас. (програмата **prog1\_2.cpp** е даденa отделно). При изпълнение на програмата се виждат следните резултати:

Обектът **ob** след извикване на функция setab(5): 5 25

Обектът **ob** след задаване на стойност на члена ob.b=20: 5 20

Обектът **ob** след извикване на функция ob.reset(): 0 0

Как се осъществява достъп до членовете на класа **myclass**? Например във член-функцията **setab** при двете присвоявания  **a = i;**  и **b = i\*i;** обръщението към членовете данни **a**  и **b** е **непосредствено**. **Това е така, защото една функция-член винаги се извиква за определен обект.**  **А кодът се изпълнява, когато тя е извикана, следователно обектът е известен.**   
Т.е. няма нужда да се използва операторът “точка”.

Променливата **b** е **public**-член на класа **myclass**. Това означава, че достъп до

нея има и кодът, определен извън тялото на класа **myclass**. Т.е**. ob.b = 20;** и **cout << ob.b;** са валидни инструкции. **Но обърнете внимание**, инструкциите не принадлежат към тялото на класа **myclass**, затова **достъпът до b е възможен само с използването на конкретен обект** (в случая **ob**) и оператора “точка”. Подобно е и извикването на функцията **reset()** от функция **main()** с инструкцията **ob.reset();**

Понеже **reset()** е открит член на класа, тя също може да се вика от код, дефиниран извън тялото на **myclass**, пак посредством конкретен обект (в примера отново обекта **ob**). Накрая да се спрем на тялото на функция **reset()** т.е. инструкция **setab(0);** Тъй като **reset()** е член на класа **myclass**, тя може **непосредствено да извика всяка друга функция на същия клас**, в случая функцията **setab(0);** със същата обосновка, с която вече доказахме верността на инструкции **a = i;**  и **b = i\*i;**

**Да обобщим отново накратко:** когато достъпът до някой член на даден клас става извън тоя клас е **необходимо да използваме името на конкретен обект**. Но **кодът** на една функция-член може да се обръща към другите членове на същия клас **напряко (непосредствено).**

**Конструктори и деструктори**

**Конструктор — това е функция, която се вика при създаване на обект**.

По правило, някои части на един обект се инициализират преди той да бъде използван. Да се върнем към класа **queue**. Тук преди всичко останало, с помощта на функцията **init()**, се присвояват нулеви стойности на променливите **rloc** и **sloc**. Но доколкото изискването за **инициализация** на членовете на един клас е доста разпространено, в C++ е предвидена реализация на тая възможност **още при създаване на обектите на класа**. Такава автоматична инициализация си изпълнява благодарение на конструкторите.

**Конструктор — това е специална функция, която е член на класа и името ѝ съвпада с името на класа.** Следва нова реализация на класа **queue** с използването на конструктор за инициализация на членовете му:

/\* Определение на класа queue\*/ **class queue { int q[100]; int sloe, rloc; public:** **queue(); /\* конструктор \*/ void qput(int i); int qget();};**

Обърнете внимание на това, че в обявлението на конструктора **queue()** отсъства **тип** на връщаната стойност. **В C++ конструкторите не връщат стойности**, следователно, няма смисъл да се посочва техния тип. (При това **не трябва** да се указва даже типът **void**.)

Сега да покажем кодът на функцията **queue().**

**/\* Определение на конструктор \*/** **queue::queue(){**

**sloe = rloc = 0;**

**cout << "Опашката е инициализирана.\n";**

**}**

**Конструкторът на обект се вика при създаване на обекта**. **Това означава, че той се извиква при изпълнение на инструкцията за обявяване на обекта**. Конструкторите на глобални обекти се викат **еднократно** в самото начало на изпълнение програмата, още преди извикването на функцията **main().**   
Що се касае за локалните обекти, то **техните конструктори се извикват всеки път,** **когато се срещне обявление на такъв обект**.

**Деструктор — това е функция, която се вика при разрушаване на обект.**

Допълнение към конструктора е деструкторът**. В много случаи при разрушаване на даден обект е необходимо да се изпълни някакво действие или даже последователност от действия.** Например, обектът е длъжен да освободи заделената по-рано за него памет**. В C++ именно деструкторът е отговорен за обработката на процеса на деактивизация на обекта.** Името на деструктора съвпада с името на конструктора, но се предхожда от символа тилда **~.** Подобно на конструкторите, **деструкторите не връщат стойности**, следователно и в техните обявления отсъства типът на връщаната стойност .

Да разгледаме програма **Prog1\_3.cpp**, дадена отделно с вече известния ни клас **queue**, но сега той съдържа конструктор и деструктор. (Тук деструктор по същество не е нужен и е включен само за илюстративни цели.)

При изпълнение на програмата се получават такива резултати:

**Опашката е инициализирана.**

**Опашката е инициализирана.**

**20 10 29 17**

**Опашката е разрушена.**

**Опашката е разрушена.**

Параметризирани конструктори (конструктори с параметри)

**Конструкторът може да има параметри**. С тяхна помощ, при създаване на обект на членовете-данни (променливите на класа) **може да се присвоят някакви начални стойности, определени в програмата.** **Това се реализира чрез предаване на аргументи на конструктора на обекта.** В следващия пример (даден отделно като **Prog1\_4.cpp**) ще усъвършенстваме класа **queue** така, че той да приема аргументи, които да служат като идентификационни номера (**ID**) на опашката. В новия вариант променливата **num** се използва за съхранение на идентификационния номер (**ID**) на създаваната от програмата опашка. Нейната действителна стойност се определя от стойността, **предавана към конструктора** като параметъра **id**, при създаването на променлива от типа **queue**. Т.е. обявяването на конструктора в рамките на класа ще изглежда така: **queue(int id); // параметризиран конструктор**

**А за да предадем аргумент към конструктора е необходимо да свържем тоя аргумент с обект** – това става при самото обявяване на обекта. **C++ поддържа два начина** за реализация на такова свързване. Ето как изгледа първият начин:

**queue а = queue(101);**

В това обявление се създава опашка с име **а**, на която се предава стойност (идентификационен номер) 101. Но тая форма се употребява рядко, доколкото вторият начин е по-удобен:

**queue а (101);**  Тая инструкция е еквивалентна на предишното обявление.

Тоест общият формат за предаване на аргументи към конструктори е:

**тип\_на\_класа име\_на\_обект(списък\_от\_аргументи);**

Елементът **списък\_от\_аргументи** представлява списък от **разделени със запетаи аргументи**, предавани към конструктора.

В примера с класа **queue** при създаване на обекта се предава само един аргумент. В общия случай е възможно предаването на два и повече аргументи.

**Забележка 1.** ***Формално*** между двете показани по-горе форми за инициализация **има малка разлика** , която ще стане ясна по-късно.

**Забележка 2.** **Само ако конструкторът има един аргумент е позволен още един начин за инициализация**. Ще го покажем с пример: В **Prog1\_4.cpp** можем да заменим **queue a(1), b(2);** със следния ред: **queue a=1, b=2;**

**Важно!** За разлика от конструкторите, **деструкторите не могат да имат параметри.** А и защо да предаваме аргументи към обект, който ще унищожим!

Класове и структури — сродни типове

Класовете и структурите са много близки. Единственото различие между **С++-**структура и **С++-**клас се състои в това, че **по подразбиране членовете на класа са закрити, а членовете на структурата - открити**. **На практика**   
**В C++ обявяването на структура създава тип клас!** (т.е. структурата е клас!)

**Понякога С++-програмистите употребяват термина POD-struct**. (Plain Old Data) **за структурите, които не съдържат функции-членове.**

Сравнение между структури и класове

Фактът, че както структурите, така и класовете имат практически идентични възможности, създава впечатление за излишество. Причината лежи в произхода на езика **C++** от **С** и намерението да се запази **C++** съвместим отдолу нагоре със **С**. В съответствие със съвременното определение на **C++,** стандартната **С**-структура представлява и напълно законна **С++-** структура.   
**В езика С**, който не съдържа ключовите думи **public** или **private**, **всички членове на структурата са открити**. **Ето защо и членовете на С++-структурите по подразбиране са открити** (а не закрити). **Понеже конструкцията от типа class е била специално разработена за поддръжка на инкапсулация, има определен смисъл в това, нейните членове да са закрити по подразбиране**. Освен това, тъй-като типът **class** синтаксично е отделен от типа **struct**, **определението на класа е напълно открито за еволюционни изменения**, които синтаксично **биха могли** да се окажат несъвместими със **С**-подобни структури. Понеже имаме работа с два отделни типа, бъдещото направление за развитие на езика **C++** не е обременено с “морални задължения”, свързани със съвместимостта със   
**С**-структурите.

Връзка между обединения и класове

Удивително, но обединенията също са свързани с “близки отношения” с класовете. **Съгласно определението едно C++ обединение по същество е също клас, в който всички членове-данни се съхраняват в една и съща област.** (**По такъв начин, обединението също дефинира тип клас**.) **Обединение може да съдържа конструктор и деструктор, а също функции-членове.** Разбира се, членовете на обединението по подразбиране са открити (**public**). Да разгледаме програма **prog1\_5.cpp** в която обединението се използва за извеждане на символите, явяващи се съдържание на старшия и младши байтове на едно цяло число (от целочислен тип).

Подобно на структурата, **С++-**обединението също произлиза от своя **С-**предшественик**. Но в C++ то има по-широки “класови” възможности.** Ако ви устройва обединение с традиционен стил на поведение, вие имате свободата именно и така да го използвате. Но в случаите, когато можете да капсулирате данните на обединението заедно с функциите, които ги обработват, все пак си струва да се възползвате от **С++-**възможностите – това ще даде на вашата програма допълнителни преимущества.

**Отново за inline функциите (пример Prog1\_5b\_inline.cpp)**

Съществува още един начин за създаване на вградена функция.   
**Той се състои в това да определим кода на функцията-член на класа в самото обявление на класа**. **Всяка функция, която е определена в обявлението на класа автоматично става вградена. В такъв случай дори не е задължително да използваме ключовата дума inline**. Например,

**class cl {**

**int i; // закрит член по подразбиране**

**public: // автоматично inline функции**

**int get\_i() { return i; }**

**void put\_i(int j) { i = j; }} ;**

**Важно! Дефинирането на неголеми функции-членове в обявлението на класа е обичайна практика в професионалното С++ програмиране!**

Масиви от обекти

**Масиви от обекти може да се създават точно така, както се създават масиви от стойности на други типове.** Например, в следващата програма **prog1\_6.cpp** се създава клас **display**, който съдържа стойности за разрешаващата способност на различни режими на монитора. В **main()** достъпът към обектите, елементи на масива, се осъществява с помощта на стандартната процедура за индексиране на масив**.**

**Обърнете внимание** на използването на **двумерния** символен масив **names** за преобразуване на изброената стойност в еквивалентен символен низ. Във всички изброявания, където отсъства явно зададена инициализация, първата константа има стойност 0, втората — стойност 1 и т.н. Следователно **стойността, връщана от функцията get\_res(),** **може да се използва за индексация на масива** **names**, което позволява да изведем на екран съответстващото название на режима на извеждане.

Инициализация на масив от обекти

Ако един клас включва параметризиран конструктор, то масив от обекти на такъв клас също може да се инициализира. Например, в следващата програма **prog1\_7.cpp** се използва параметризиран конструктор **samp** и инициализируем масив **sampArray** за обектите на тоя клас.

В действителност, синтаксисът за инициализация на масива, зададен с инструкцията **samp sampArray[4] = { -1, -2, -3, -4 };**

представлява съкратен вариант на следния дълъг формат:

**samp sampArray[4] = { samp(-1), samp(-2), samp(-3), samp(-4) };**

**Краткият формат е валиден само за масиви от такива обекти, конструкторите на които приемат само един аргумент!**

**Указатели към обекти**

Както е известно, достъп до структура може да се получи пряко или чрез указател към тая структура. Аналогично може да се обръщаме и към обект: непосредствено (както във всички предишни примери) или с помощта на указател към обект. За да получим достъп до отделен член на обекта се използва оператор “точка”. Но ако за достъп се използва указател към този обект е необходимо да се използва операторът “стрелка”. (Прилагането на операторите “точка” и “стрелка” за обекти **съответства** на прилагането им за структури и обединения.) **За да обявим указател към обект се използва същият синтаксис, както и в случая на обявяване на указатели към стойности на други типове**.

В следващия пример **prog1\_8.cpp** е показано как се използват операторите „**.**“ и „**->**“, а също така и как работят инкрементиране и декрементиране с указатели.

Да припомним, че при инкрементиране или декрементиране на **указател той се инкрементира или декрементира така,** **че винаги да указва към следващия или предишния елемент от основния тип**. **Същото се случва и при инкремент или декремент на указателя към обект.**

Както ще бъде показано по-нататък, указателите към обекти играят главна роля в реализацията на един от най-важните принцип на **C++,** а именно полиморфизма.

Псевдоним (Алтернативно име) на обект

**Алтернативни имена (псевдоними) на обекти могат да се използват по същия начин, както и при стойности от всеки друг тип**. **За тази цел не съществуват никакви специални инструкции или ограничения**. Както ще бъде показано, използването на псевдоними на обекти позволява да се решат някои специфични проблеми, които могат да възникнат при използването на класове.

Край на лекция 1.