1. Обекти и класове. Дефиниция на клас. Общи понятия и концепции.
Обектът представя нещата от реалния свят или проблем от дадена сфера (пр. Червена кола със счупения мигач), представен логически на компютър.
Класът представя всички обекти от даден тип (пр. Кола). Той представлява описание на тип, включващ едновременно данни и функции, които ги обработват. Данните се наричат член-променливи, а функциите – член-функции.
Дефиницията на един клас включва декларация на класа и дефиниции на член-функциите.
Декларацията на класа има следния синтаксис:
class <име-на-клас> {
// Декларации на член-променливи
// Декларации на член-функции };
Дефиницията на член-функция има следния синтаксис:
<тип-връщан-резултат> <име-на-клас>::<име-на-функция>(<списък с параметри>)
{ // Тяло на функцията }

За да се създаде обект, се използва името на класа като спецификатор за типа. Следният ред декларира обект ob1 от тип triangle:
triangle ob1;
Нивата на достъп са private, protected, public. При private членовете са видими само в същия класа, а при public са видими за всички.
Достъпът до public компонентите на даден обект се осъществява по 2 начина – чрез името на обекта и името на компонентата разделени с точка или чрез указател към обекта и името на компонентата:

```
ob1.face();
ob1.show(”triangle1”);
p->face();
p->show(”triangle2”);
```

2. Методи и параметри. Даннови членове. Инстанция на обект.
Функциите, членове на един клас се наричат методи(член-функции).
Обектите имат операции, които могат да бъдат извиквани наречени методи. Методите могат да имат параметри, които подават допълнителна информация нужна за изпълнение.
Инстанцията на обект е заделен в паметта обект от даден тип клас. Много инстанции могат да бъдат създадени от отделен клас.
Обектът има атрибути, чиято стойност се записват в полета. Класът дефинира какви полета да има обекта, но всеки обект съдържа свой, заделени в паметта, множество стойности за съответните полета. (състоянието на обекта).
class <име-на-клас> {
// Декларации на член-променливи
// Декларации на член-функции };
Дефиницията на член-функция има следния синтаксис:
<тип-връщан-резултат> <име-на-клас>::<име-на-функция>(<списък с параметри>)
{ // Тяло на функцията }

3. Класове и обекти. Преход от структура към клас.

Класът представлява описание на тип, включващ едновременно данни и функции, които ги обработват. Данните се наричат **член-променливи**, а функциите – **член-функции**.
Обекти са заделени в паметта единици от даден тип клас.
Класът синтактически прилича на структурата. Класовете и структурите имат идентични потенциални възможности. В C++ деф на една структура е разширена така, че да включва член функции включително конструктури и деструктури по същия начин както и класовете. Единствената разлика м/у структура и клас е, че по подразбиране членовете на един клас са Private, а тези на структурата – Public.
Разширен синтаксис на една структура:
struct <име-на-тип> {
    //public функции и променливи
  private :
    //private функции и променливи
} <списък-от-обекти>;

4. Конструктори
може да се изпълни автоматично от функцията конструктор.
Конструкторът има същото име като името на класа, към който принадлежи, и не притежава тип на връщан резултат. Един клас може да има повече от един конструктор, с различни аргументи. Ако в класът липсва дефиниран конструктор, то той се добавя автоматично от компилатора, като той само заделя памет за член-променливите без никаква инициализация, и няма параметри.. Общият вид на конструктора е:
<име-на-клас>::<име-на-клас>(<списък формални аргументи>)
{ //Тяло на конструктор}
Конструкторите не се наследяват автоматично и трябва ръчно да се предефинират при наследени класове.

5. Деструктори
Деструкторът се извиква автоматично при разрушаването на обект от класа.
Деструкторът има същото име като класа, но предшествано от символа ~. Деструкторът не може да връща резултат и не може да има аргументи. В един клас може да има само един деструктор. Ако липсва, то той бива генериран автоматично от компилатора, като в него просто се освобождават член-променливите на класа. Общият вид на деструктора е:
<име-на-клас>:: ~<име-на-клас>()
{ //Тяло на деструктор }
Деструкторите не се наследяват автоматично и трябва ръчно да се предефинират при наследени класове.
При наследяване е добре да дефинираме базовия деструктор като виртуален за да не възникнат проблеми при освобождаването на памет.

6.Типове променливи и оператори в клас.Основни типове данни.
В C++ има различни типове променливи – за цели числа, за числа с десетична запетая, за символи, низове и тн. Примери за типове променливи са : char, short int, int, float, double , bool и тн.

Оператори: В езика C++ има богат набор от оператори.В него са дадени също и средства за предефиниране.
Всеки оператор се характеризира с :

-позиция на оператора спрямо аргомента му;
-приоритет
-асоциативност;
Позицията на оператора спрямо аргумента му го определя като: префиксен(оператора е пред единствения си аргумент --i), инфиксен(оператора е между аргументите си-4+2),постфиксен(оператора е след аргументите си i++).
Приоритетът на оператора определя редът на изпалнение на операторите.Оператора с по-висок преоритет се изпалнявя преди оператора с по-нисак преоритет. (Пример: Преоритета на * и / е по-висок от този на + и -).
Асоциативността определя реда на изпълнение на операторите с еднакъв преоритет. В C++ има ляво асоциативни и дясно асоциативни оператори. Ляво асоциативните оператори се изпалняват от ляво на дясно, а дясните - от дясно на ляво. В C++ не може да се дефинират нови оператори, но всеки съществуващ вече оператор с изключение на **::, . , ?:** и **.*** може да бъде предефиниран от програмиста, стига поне един операнд на оператора да е обект на някакъв клас.
За предефинирането на оператор се създава операторна функция. Най-често операторната функция е член-функция или приятелска функция на класа, за който е дефинирана.Съществуват две важни ограничения при предефинирането на оператор. Първо, приоритетът на операторите не може да бъде променян. Второ, броят на операндите, които приема този оператор, не може да бъде променян.
Операторите биват : за присвояване(=), бинарни(+ , - и тн), релационни и логически (примерно == и &&), унарни (пример: ++), оператор за индекс ([ ]) и др .

Основни типове данни
- Цели числа (int: 2-4 байта и лонг минимум 4 байта): 8, -54, 1289
- Дробни числа с плаваща – float или фиксирана точка - double: 0.01, 12.5, 73.0, 21.0E2
- Символни имена (идентификатори):

Аритметични операции в C++.
+/- - събиране и изваждане
*,/,% - умножение, деление, остатък от деление
Особености:
- Приоритети - както в математиката (най-напред се извършват действия *, / и % и след тях + и -)
- Използване на скоби за промяна на реда на изпълнение на аритметичните операции.
- Аргументна зависимост - действието на операцията зависи от типа на аргументите.
- Двата аргумента на аритметичните операции трябва да са от един и същи числов тип данни; резултатът е от същия тип.
- Разрешено е единият от аргументите да е от тип double/float, а другия - от тип int; тогава аргумента от тип int се преобразува (автоматично) на тип double и след това се извършва аритметичната операция - с аргументи тип double.

7. Указатели и референции. Константи. Изброими типове.
Указател е променлива съдържаща адреса на даден обект или променлива, а не самият той. Указателят трябва да бъде от същия тип (или производен), какъвто е обекта, когото сочи. Пример:
int X=10; – декларира променлива X със стойност 10
int* pX= &X; – декларира указател и му дава адреса на променливата X
*pX = 4; - достъпване на самата стойност която сочи указателят.

Референциите позволяват да се създаде 2ро име на дадена променлива, което се използва за да се променя оригиналът. Ако аргументите на дадена функция са референции, то тогава може ефективно да се променя информацията подадена на функцията. Референцията е нещо като постоянен указател, самата референция не може да се променя без да повлияе на оригиналът, т.е. същия обект, второ име.
Пример:
Int X=10; --променливата = 10
Int& Y=X; --създаване на референция на X
Y=4 --промяна на стойността на X чрез У
X и Y са една и съща променлива/обект, т.е. (&X = = &Y) ще бъде true.

Константите се използват когато трябва параметър, който няма да бъде променян от програмата. Константите могат да бъдат задавани по два начина:

```
#define PI 3.14159;
#define NEWLINE '\n';
const char tabulator = '\t';
const char *Function1() { return "Some text";};
```

Когато указател е деклариран като константа, то обектът когото сочи става константа, т.е. не може да се променя:

```
const char* p = ”abcd”;
p[3] = ‘a’; // wrong!
p = “qrtw”  //OK!
```

Ако искаме указателят да е константен и да сочи постоянно даден адрес:

```
char* const p = “abcd”;
p[3] = ‘a’; // OK!
p = “qrtw”; //wrong!
```

Възможно е двата варианта да се комбинират:

```
const char* const p = “abcd”;
```

Изброимите типове:
Изброими типове в C++ са: bool, char, int и т.н.
Изброим тип е крайно множество от стойности. В C++ всяка единица от това множество съответства на int константа. Дефинират се чрез enum:

```
enum WorkDays {Monday, Tuesday, Wenesday, Thursday, Friday}
WorkDays today = Monday;
```

today е променлива от изброения тип и може да й се присвояват само стойности от множеството. Възможно е да се зададе на коя единица какво число да отговаря:

```
enum WorkDays {Monday=1, Tuesday=2, Wenesday=3, Thursday=4, Friday=5}
```

8. Предаване на данни като параметри. Accessor методи. Mutator метод.
Във класовете, добър стил на програмиране, е полетата да бъдат private за да не са достъпни отвън, с цел защита от неправомерно променяне на данните в класа. За тази цел се правят допълнителни public методи, чрез които достъпът до полетата става възможност и така класът се капсулира. Тези методи се наричат Accessor/Mutator (get/set).

Accessor (get) методите често нямат параметри и служат за прочитане на дадено поле или набор от данни в класа – read only. Обикновенно връщат директно стойността на полето. За тази цел типът на стойността която връща функцията трябва да съответства на типът полето. Така макар че полето е private, може да се вземе неговата стойност отвън класа, а при липсата на такъв метод, може да бъде скрита.

```
float getPrice() {
  return price;
```

```
};
```
Mutator (set) методи често имат 1 или повече аргументи и служат за променяне на стойността на 1 или повече полета – write. Обикновенно тези методи не връщат стойност (са void тип). Типът на аргументите трябва да съответства с типовете на полетата с които ще работиме. По този начин макар че полето е private, може да променим неговата стойност при нужда. Използването на Mutator методи ни позволява да сложим проверки за валидност на подаваните данните при променяне на полетата.
```
void setPrice(float pr) {
  if (pr>0.0) price=pr;
};
```

9. Общи понятия:функции; декларация на прототип; дефиниране на функция; повикване на функция; работа с локални и глобални типове; дефиниране и използване на overloaded функции.

Функция на C++ е начин да се разделят блоковете код на части. Чрез тях програмата може да раздели така, че да не трябва непрекъснато да се пише един и същи код, а просто да се извика функцията. Чрез функциите кода се поддържа малък, чист и функционален. Всяка функция има три части — прототип (незадължителен, но препоръчителен), заглавна част и тяло.

Прототипът указва на компилатора, че определена функция съществува, но тялото на функцията е някъде другаде. Той може да бъде пропуснат, като се сложи заглавната част и тялото над всички извиквания на функцията, но в някои случаи това е невъзможно. Освен това, прототипа помага за изчистването и разбираемостта на кода, което е и главната цел на функциите. Прототипът има следния синтаксис:
<връщан тип> <име> ([тип] [параметър1], [тип] [параметър2],...);

Заглавната част се пише непосредствено преди тялото на функцията, за да се укаже на компилатора коя функция се описва. Заглавната част има същия синтаксис като прототипа, но без ';' накрая. Например:
double Area(double Width, double Height)

Тялото на функцията е този код, който всъщност се изпълнява. Тялото започва с отворена фигурна скоба и завършва със затворена фигурна скоба ({ и }). Между тях трябва да се напише кода на функцията. Ето например реализацията на функцията Area:
```
double Area(double Width, double Height)
{
 double Ar=Width*Height;
  return Ar;
}
```
Чрез return се указва какво връща функцията, в случая лицето на правоъгълник с размери Width и Height. Тоест, ако се напише в main или в някоя друга функция следното:
cout<<Area(5,10)<<endl; на екрана ще се изпише 50.

Локалните променливи могат да бъдат променливи от всички стандартни и изброени типове в програмата. Те са локални за самата част на програмата (тялото на функцията или класа). Създават се когато тази част се извика и се унищожават веднага след приключване на работата на тази част.

Глобалните променливи са променливи, които се дефинират преди дори прототипите на функциите и тази глобални променливи могат да се използват във всяка част на програмата. Те се създават в началото на програмата и се унишожават веднага след приключването й.

В C++ две или повече функции могат да имат едно и също име, като за това е достатъчно или типът на техните аргументи да се различава, или броят на аргументите

им да е различен, или и двете. Когато две или повече функции имат едно и също име, за тях казваме че са **предефинирани** или **overload**. Не може да се предефинира функция ако тя се различава само по връщащия тип.

10. Функции – методи. Глобални функции

Функция на C++ е начин да се разделят блоковете код на части. Те предоставят на програмиста начин да раздели програмата си така, че да не трябва непрекъснато да се пише един и същи код, а просто да извика функцията. Чрез функциите кода се поддържа малък, чист и функционален.

Всяка функция има три части — прототип (незадължителен, но препоръчителен), заглавна част и тяло.

Прототипът указва на компилатора, че определена функция съществува, но тялото на функцията е някъде другаде. Бихте могли и да не пишете прототип, като сложите заглавната част и тялото над всички извиквания на функцията, но в някои случаи това е невъзможно. Освен това, прототипа помага за изчистването и разбираемостта на кода, което е и главната цел на функциите. Прототипът има следния синтаксис:

<връщан тип> <име> ([тип] [параметър1], [тип] [параметър2],...);

Връщания тип може да бъде всеки тип променлива или void. Името на функцията трябва да бъде съставено от символите A-Z, a-z, 0-9 и _ (долна черта), като не може да започва с цифра. Името трябва да пояснява какво прави функ., да не е прекалено дълго, нито прекалено късо. Последната част от прототипа е опис на параметрите. Това е списък със стойности, които се подават на функцията. Те могат да бъдат колкото поискате (разделят се със запетаи), или може изобщо функцията да не приема параметри. Ето няколко примера за прототипи:

```
double Square(double Number);
void ShowHelp();
double Area(double Width, double Height);
```

За да извикате някоя от тези функции, напишете например

```
Square(5); ShowHelp(); Area(5,10);
```

Заглавната част се пише непосредствено преди тялото на функцията, за да укажете на компилатора коя функция ще описвате. Заглавната част има същия синтаксис като прототипа, но без ';' накрая. Например

double Area(double Width, double Height)

Тялото на функцията е този код, който всъщност се изпълнява; то е това, което прави функцията. Тялото започва с отворена фигурна скоба и завършва със затворена фигурна скоба ({ и }). Между тях трябва да напишете кода на функцията. Ето например реализацията на функцията Area:

```
double Area(double Width, double Height) {
 double Ar=Width*Height;
 return Ar;
}
```

Чрез return указвате какво връща функцията, в случая лицето на правоъгълник с размери Width и Height. Тоест, ако напишете в main или в някоя друга функция следното:

`cout<<Area(5,10)<<endl;` - на екрана ще се изпише 50.

11. Припокриване на функции (overloading)

Езикът C++ позволява функции с едно и също име да имат различно съдържание и действие. При обръщение към функция с дадено име се зарежда тази от едноименните функции, която отговаря най-точно по брой, тип и ред на аргументите. Не могат да съществуват две функции или повече, които се различават само по тип на връщания

резултат. Трябва или параметрите да са различни по брой, или по подредба или типовете в дадената последователност (т.е. да имат различни сигнатури). Достатъчно е един от параметрите да е с различен тип.

int average(**double** nmber1, **double** number2);
int average(**int** array[], **int** arraysize);

12. Локални променливи. Локални и/или глобални елементи на клас.
Локални променливи са променливи които имат живот само във блока в който са създадени, примерно в блока на даден метод на класа. В класовете, това са полетата му-променливи които са видими за дадения клас (private) или за всички (public). Тези полета се създават и разрушават заедно със обекта от дадения клас. Полета/методи които са protected биват видими за дадения клас и всеки клас който го наследява.
По подразбиране класа си прави полетат private.
Глобални елементи за даден клас е всичко което е декларирано извън класът и какъвто и да е друг блок/функция.

13.Взаимодействия на обекти.Понятие за абстракция и модулност.
Абстракция - Способност на програма да игнорира някои аспекти на информацията, която манипулира, т.е. възможността да се концентрира над основните проблеми. Всеки обект в системата служи като модел за абстрактен "актьор", който може да върши определена работа, да променя състоянието си, да дава информация за себе си и да "комуникира" с други обекти в системата, без да разкрива как точно са имплементирани свойствата му. Процеси, функции и методи също могат да бъдат абстрактни, и когато са такива, редица техники са нужни за да се разшири абстракцията.
Модулността позволява на части от програмата да съответстват на отделни аспекти на проблема.
Така може да включиме цели модули със различни функционалности, и да ги използваме без да се интересуваме как се изпълнява тази функционалност, ами само от крайният резултат.

17. Оценка на качеството на кода. Понятия и оценка за свързаност и структурираност.
Когато се пише една програма трябва да винаги да пишем код който лесно да може лесно да се поддържа и променя. За тази цел, освен прегледността и подредбата на кода, трябва да се съобразяваме с понятията свързаност (coupling) и структорираност (cohesion) за да имаме качествен код.
Свързаност имаме когато имаме връзки м/у отделни компоненти на една програма. Когато 2 класа имат много връзки един с друг (използват взаимно методите си), то те са силно свързани. Това води до трудно проследяване или промяна на кода без да се повлияе на други класове. При такива случаи може да се проектира по добре като слеем 2-та класа в един. Цели се ниска свързаност, което позволява лесна поддръжка.
Структорираност означава целенасочеността на отделните компоненти на една програма. Един компонент (клас или метод) трябва да изпълнява само една задача за която е предназначен. В такъв случай се казва че има висока структорираност. Ако един метод прави повече от една задача, то по добре е да създадем 2 метода които да изпълняват 2-те задачи по отделно. Това спомага за универсалността на компонента, както и лесното й прочитане и поддържане.

18. Оценяване на кода: дублиращи се фрагменти. Целево-оиентиран проект. Пример.

Дублиращ се код в програма обозначава лош дизайн/структура на кода и прави поддръжката му трудна. Когато е нужна промяна във кода, то тази промяна трябва да се направи на всяко копие на този код. А самият код става дълъг и неоптимизиран. За оптимизиране може да се ползват изброен тип или като се разбие на по-малки методи, правейки кода по гъвкав, сбит и прегледен.

Пример:

//keys е масив който съдържа всички клавиши дали са натиснати или не.

```
void KeysPressed(bool keys[]) {
  cout << "Pressed keys are: \n";
  if (keys[VK_ENTER]) cout << "Pressed Enter \n";
  if (keys[VK_LEFT]) cout << "Pressed Left \n";
  if (keys[VK_UP]) cout << "Pressed UP \n";
  … //Още клавиши
  if (keys['A']) cout << "Pressed A \n";
  if (keys['B']) cout << "Pressed B \n";
  … //Още букви
}
```

Ако в този пример се опитаме да направим превод на български, трябваше на всеки ред да променим “Pressed “ със “Натиснат ”.

Пример- оправен:

```
void KeysPressed(bool keys[]) {
  cout << "Pressed keys are: ";
  for(int i; i<= KEYSMAX; i++)
     if (keys[i])  //Ако i-тият клавиш е натиснат
        cout << „Pressed " << getKeyName(i) << „ \n";
}
```

За да се избегне повтарянето на цели методи в сходни класове се използва наследяването на класове, където от даден базов клас се наследяват всичките функции и полета, и няма нужда да се имплементират наново в новият ни клас, което ни позволява да опишеме само нещата по които се различава новия клас от базовия.

19. Проблеми в проектиране и ООП. Водещи практики.

Главни проблеми при ООП програмирането са: структорираност (cohesion), свързаност (coupling) и дублирането на код.

При структорираността, проблемът е всеки метод да изпълнява само една функция, за което е предназначена. Когато става въпрос за класове, те трябва да бъдат ясно разграничими и дефинирани единици, удволетворяващ нуждата от него.

При свързаността, имаме класове които се използват взаимно. Ако има прекалено много връзки между 2 класа, това говори за лош дизайн на кодът.

При дублирането на код се усложнява поддръжката на кода и говори за лош дизайн.

Добрият дизайн включва:

Разпределяне на отделните методи в подходящи класове.

Всеки клас трябва сам да манипулира свойте данни, както и да се грижи за валидността им.

Ниската свързаност води до локализация на промените- ако е нужна промяна трябва да засегне колкото се може по малко класове. Ако промяната е трудна – лош дизайн.

Good code: no duplication, high cohesion, low coupling.
Един метод е твърде дълъг когато изпълнява повече от една функция.
Един клас е твърде сложен когато изпълнява повече от 1 логическа цел.

20. Класове и обекти: декларация и дефиниция. Създаване и унищожаване обекти
Класът представлява описание на тип, включващ едновременно данни и функции (които ги обработват). Класовете могат да се наследяват. Данните се наричат **член-променливи** или **полета**, а функциите – **член-функции** или **методи**.
Дефиницията на един клас включва *декларация на класа* и полетата и методите.
Класът е шаблон, който се използва за създаване на обекти. Един клас се декларира посредством ключовата дума class.Синтаксисът на една декларация на клас е сходен с тази на една структура:

```
class <име-на-клас> {
    private:   //незадължителен
        //private променливи
    public:
        //public функции и променливи
} <списък от обекти>;
```

В декларацията на класа обектите могат да се декларират и по-късно.Функциите и променливите в даден клас са негови членове. По default те са private членове на класа и са достъпни само в неговите рамки. За да се декларират публични членове е необходимо да се използва ключовата дума „public:”.По принцип за дефинирането на член-функция се използва следната форма:

```
<тип-връщан-резултат> <име-на-клас>::<име-на-функция>(<списък –параметри>)
{
  //тяло на функцията
}
```

Декларацията на клас е логиеска абстракция, която дефинира нов тип. Тя определя как ще изглежда един обект от този тип. Декларацията на обект създава физическа единица от такъв тип. Тоест, един обект полетата му заемат памет и методите му могат да се извикат, докато на класът-не. Всеки обект от даден клас притежава свое собствено копие от всяка променлива, декларирана в този клас.

За създаване на обект се използва следната декларация

```
int main() {
   mysclass ob;
   return 0;
};
```

Обектът, при създаването си може да приема различни по типове промеливи декларирани в конструктура на базовия клас. Броят им трябва да съотвества на броят, който е деклариран в конструктура.
Обектът освобождава паметта когато излезе извън обсега на блока или когато бъде ръчно освободен чрез delete (ако е бил заделен с new), при което се извиква неговият деструктор.
Един обект може да бъде присвоен на друг обект, като полетата на единия се копират в полетата на другия. Трябва да се внимава в случаите когато има полета-указатели и освобождаването на обектите които сочат.

21. Дефиниране на конструктори и деструктори. Дефиниране на методи на класа.
Конструктурът на един клас се извиква всеки път, когато се създава обект от този клас. И поради това всякаква инициализация, която трябва да се извърши за даден обект, може да се изпълни автоматично от функцията конструктор.
Конструкторът има същото име като името на класа, към който принадлежи, и не притежава тип на връщан резултат. Един клас може да има повече от един конструктор, с различни аргументи. Ако в класът липсва дефиниран конструктор, то той се добавя автоматично от компилатора, като той само заделя памет за член-променливите без никаква инициализация, и без параметри. Общият вид на конструктора е:
<име-на-клас>::<име-на-клас>(<списък формални аргументи>)
{ //Тяло на конструктор}

Деструкторът се извиква автоматично при разрушаването на обект от класа.
Деструктора служи главно за освобождава заделената за обекта динамична памет.
Деструкторът има същото име като класа, но предшествано от символа ~. Деструкторът не може да връща резултат и не може да има аргументи. В един клас може да има само един деструктор. Ако липсва, то той бива генериран автоматично от компилатора, като в него просто се освобождават член-променливите на класа. Общият вид на деструктора е:
<име-на-клас>:: ~<име-на-клас>()
{ //Тяло на деструктор }

Конструкторите и деструкторите не се наследяват автоматично и трябва ръчно да се предефинират при наследени класове.

Функциите, членове на един клас се наричат методи(член-функции).
Дефиницията на член-функция има следния синтаксис:
<тип-връщан-резултат> <име-на-клас>::<име-на-функция>(<списък с параметри>)
{ // Тяло на функцията }

22. Дефиниране на връзки.
Връзките между класовете показват какви са взаимоотношенията между 2 класа. Има 2 типа връзки в ООП: “is-a” и “has-a”.
“is-a” е когато един клас наследява друг клас. Тоест наследеният клас има същата функционалност и структора като базовият клас, като е възможно да го разшири.
Пример: apple наследява fruit. Тогава може да кажем: apple IS A fruit.
“has-a” е когато в един клас се съдържат членове на друг клас. Когат се ползва такава връзка, обикновенно трябва ръчно да заделим този допълнителен обект в паметта, след което не трябва да се забравя да се изчисти паметта от него. Честа грешка е да се изтрива главния обект, без заделените от него съдържани обекти -> memory leak.

23. Наследяемост. Подтипове. Заместявания. Полиморфизъм и променливи.
Наследяване в ООП наричаме възможността един клас, наричан **наследник**, да придобие свойства и действия на друг клас – **родител** (**базов** клас).
Наследяемост:
- Споделянето на атрибути и операции между класовете, базиращо се на йерархична връзка
- Класовете могат да се дефинират по-общо и после да се конкретизират в по-точни подкласове

- Всеки подклас включва или наследява (inherit) всички свойства на родителския си клас (super class) и добавя свои уникални свойства

Полиморфизъм (polymorphism) е способността на един обект да съществува в различни форми. Полиморфизмът дава възможност да използвате един и същ метод за изпълнението на различни задания. В производните класове можете да променяте реализацията на метод от базовия клас. Следователно, когато от един базов клас създавате два производни класа, можете да създадете метод с едно и също име и в двата класа. Но методът във всеки от производните класове изпълнява различни задачи. Избира се кой метод да се извика в зависимост от това от кой обект се вика. Пример:

```
Animal *ptr = new Cat(…);
ptr->speak(); //мяу
ptr = new Dog(…);
ptr->speak(); //бау;
```

24. Програмни практики: БД и разнородни обекти в нея. Проект, обектен модел, класова диаграма, реализация в код.

Проектира се база данни в която се съдържа информация за продуктите. Отделните видове продукти са отделни класове: Audio CD, Video, а техните private полета съдържат информацията за дадения продукт.
CD: title, artist, tracks, playtime, gotit; **Video**: title, director, playtime, gotit;
Създават се съответните get/set методи за работа с полетата. Пример:

```
class CD {
  private:
    char * title;
    char * artist;
    int tracks;
    int playtime;
    bool gotit;
  public:
    char * getTitle();
    void setTitle(char * value);
    …
}
```

След което в програмата ни се създава масив(база от данни)от инстанции от тези класове:
CD CDList[10]; Video VideoList[10];
При което може да се обхождат тези масиви и да се обработват отделните инстанции. Вижда се че 2-та класа са доста сходни, и доста от методите им и полетата се повтарят-получава се дублиране на код, и прави поддръжката/промяната на кода по трудна.

25. Програмни практики: БД и разнородни обекти в нея – подобрена версия с наследяемост. Суперклас и подкласове. Наследяемост.

Проектира се база данни в която се съдържа информация за продуктите. Създава се клас Item, в който се съдържат общите полета/методи на продуктите. Той е базов клас. Дефинират се 2 подкласа наследяващи Item: CD и Video и съответно неговите полета и методи. Допълнително, CD и Video разширяват наследеният базов клас Item, със нови поелта/методи специфични само за тях. Пример:

```
class Item {
  private:
    char * title;
    int playtime;
    bool gotit;
```

```
  public:
    char * getTitle();
    void setTitle(char * value);
    …
}
class CD: public Item {
  private:
    char * artist;
    int tracks;
  public:
    char * getArtist();
    void setArtist(char * value);
    …
}
```

След което в програмата ни може да се създаде масив(база от данни) от инстанции на тези класове: CD CDList[10]; Video VideoList[10];
А може и да се създаде един масив, общ за 2-та вида продукти: Item *items[10] в който могат да се съхраняват CD и Video инстанции: items[0] = new CD(…); items[1] = new Video(…);
При наследяването се избягва дублирането на код и прави поддръжката му по лесна и лесно може да бъде добавен нов вид продукт.

27. Подтипове, подкласове и присвоявания. Подтипове и предаване на параметри.
Класовете дефинират тип, а подкласовете дефинират подтип. Базовият клас се явява главен и той може да бъде оприличен на всеки един от наследяващите го класове, т.е. е логически свързан със него и практически наследеният клас може да го замести в даден момент.
Пример: имаме база от данни (масив) от указатели към обекти Item-и. Класът Item е общ за всичките разновидности обекти които може да има в базата данни.

```
Item *db[10];
db[0] = new Video(…);
db[1] = new CD(…);
db[2] = new DVD(…);
db[3] = new Video(…);
```

Тук всички новосъздадени обекти се падат под-тип на Item.

28.Полиморфични променливи.
Полиморфични са променливи, които държат обект и могат да държат повече от един тип обект, подтип на типа на самата променлива.
Пример:

```
Animal a(…);
Cat b(…);
a=b; //Ще работи.
```

Така полиморфизъмът се запазва. Но такива обекти-променливи ще имат живот само до края на блокът в който са били инстанцирани.

29. Наследяемост в ООП . Дефиниране на йерархията. Достъп до методи и данни на различни нива
Характерно за наследяемостта е, че полетата и методите (без private) се копрат и в наследяващия клас, което предотвратява повторното писане на телата им в наследяващите класове.
Когато един клас наследява друг се използва обшата форма:
<class име-производен-клас> : <тип-достъп> <име-базов-клас>
{ … };

Тук тип-достъп е една от трите ключови думи: public, private или protected.
Спецификаторът за достъп определя по какъв начин елементите на базовия клас се наследяват от производния клас. Когато спецификаторът е public, всички public членове на базовия клас стават public членове и на производния клас. Ако спецификаторът е private, всички public членове на базовия клас стават private за производния клас. И в двата случая всички private членове на базовия клас си остават private за него и не са достъпни от производния клас.
Спецификаторът за достъп protected е еквивалентен на спецификатора private с едно единствено изключение – protected членовете на базовия клас са достъпни за членове на всеки на всеки клас, който е производен на базовия клас. Извън базовия клас или неговите производни, protected членовете са недостъпни. Когатоедин protected член на клас се наследи като public от производен клас, той става protected член на производния клас. Ако базовия клас се наследи като private, един protected член на базовия клас става private член на производния клас. Един базов клас може да бъде наследен и като protected от един производен клас. В такъв случай public и protected членовете на базовия клас стават protected членове на производния клас.
Ако отсъства спецификатор, то по подразбиране той е private за клас и public за структора.
Тъй като конструкторът не се наследява автоматично, трябва да го предефинираме наново или да го наследим от базовия клас. Синтаксисът за наследяване е:
<производен-конструктор>(<списък-аргументи>): <базов-клас>(<списък-аргументи>)
{ …}
Същото се отнася и за деструкторите.

Съществуват два начина, по които един производен клас може да наследи повече от един базов клас. Първо, един производен клас може да бъде използван като базов за друг производен клас, като по този начин се създава многостепенна класова йерархия. В този случай за оригиналния базов клас се казва, че е индиректен базов клас на втория производен клас. Второ, един производен клас може директно да наследява повече от един базов клас. В този случай два или повече базови класа се използват за създаването на производен клас.
Когато един производен клас директно наследява множество базови класове, се изреждат със запетайки така:
class <име-производен-клас>: <тип-достъп> <база1>, <тип-достъп> <база2>, ... N
{ ... };

В случая когато производния клас наследява директно множество базови класове е възможно да възникне проблем, а именно да се наследи два или повече пъти даден базов клас, намиращ се по-нагоре в йерархията. Проблемът може да се реши чрез наследяване на базовия клас като виртуален. Това предотвратява присъствието на две или повече копия на базовия клас във всеки следващ индиректен производен клас. Когато се създава подобен производен клас, ключовата дума virtual се поставя пред спецификатора за достъп до базовия клас.

30. Виртуални елементи и реализация на полиморфизма. Абстрактни класове и абстрактни методи

Виртуални членове

Виртуален метод се нарича предефиниран метод, който се извиква от наследен клас, когато се обръщаме за него като към базов клас. По правило за да декларираме даден член като виртуален, трябва декларацията да е предшествана от ключовата дума virtual:

```
class myclass {
     …
  public:
      …
      virtual int area () { return (0); }
  };
```

Член функцията area() е декларирана като виртуална в базовия клас защото по-късно се предифинира в наследения клас.

Следователно virtual е разрешаването на члена от наследеният клас със същото име като в базвия клас, да бъде подходящо извикан когато чрез указател/референция се обръщаме към него като към базов клас. Когато указател тип базов сочи към обект от наследен клас, ще се извика метода на наследения а не на базовия клас. Пример:

```
Animal *ptr = new Cat();
ptr->speak();
```

Ще се извика speak() методът на наследеният клас, а не на базовия както би станало без virtual.

Клас, който е деклариран или наследен от виртуална функция се нарича полиморфичен клас.

Абстрактни базови класове

Абстрактните класове са подобни на нормалните класове, с разликата че те съдържат методи които нямат дефинирано тяло в себе си. Това става с добавяне на "=0" в декларацията на виртуалния метод:

```
class myclass{
     …
  public:
     …
    virtual int area () =0;
};
```

Всички класове съдържащи най-малко една абстрактна функция са абстрактни класове.

Основната разлика е, че в абстрактният клас ако най-малко за един от тези членове липсва имплементация, ние не можем да създадем инстанция(обект) от този клас.

Но клас ,от който не може да се инстанцира обект, не е изцяло неизползваем. Можем да създадем указатели и да ги използваме напълно във всички полиморфични "ситуации". Пример:

```
myclass poly; //НЕ МОЖЕ
myclass *poly1=new Rectangle(10,5);
```

Първият ред няма да сработи защото незнае какво е тялото на абстрактните методи и не може инстанцира обект от класа. Иначе, указателите от този абстрактен клас, могат да сочат не-абстрактни обекти от наследените класове.

Виртуалните членове на абстрактните класове дават на C++ полиморфични характеристики, така че обектно-ориентираното програмиране да се използва пълноценно в големи проекти.

31. Програмни практики: наследяемост и поломорфизъм в програма.

Основна характеристика на C++ е наследяването. То позволява да се създават класове, получени от други класове, така че те автомачино включват някои от свойте „родителски” членове, както и свойте собствени.
Когато имаме много общи полета за различни обекти се използва наследяване. Пример: триъгълник и правоъгълник, общите неща са страна и височина, защото са подклас на полигон. Когато един клас наследява друг се използва следната форма.
class <име-производен-клас>: <тип-достъп> <име-базов-клас> {
  //член-променливи специфични за този клас
  //член-функции специфични за този клас
};
Тук тип-достъп е една от трите ключови думи: public, private или protected. Ако базовият клас е деклариран като public в производния клас, всички public, private и protected компоненти на базовия клас се наследяват съответно като public, private и protected компоненти на производния клас. При private е по-разично. Когато клас наследява друг клас като private то той наследява всички public членове като private.

Не се наследяват конструктори, деструктори и приятелски функции.
Тъй като конст. не се наследяват автоматично и трябва да се предефинират наново, като аргументите на конструкторът от наследеният клас трябва съвпадат или разсширяват аргументите на конст. от базовия клас. Един конструктор може да наследи базов конструктор така:
<производен-конструктор>(<списък аргументи>):<базов-клас>(<списък аргументи>) {
  //тяло на конструктора на производния клас
}

Полиморфизъм-способността на обекти, принадлежащи към различни класове(типове) да изпълняват метод извикан с еднакво име, всеки според подходящия начин.
Операторът overloading на цифрови оператори =,-,*,/ позволява полиморфична обработка на различни числени типове: int,double т.н. всеки, от които има различни диапазони и представяне.
Полиморфизмът се използва при наследяването на класове. При тях е много подходящо общите методи да бъдат с еднакви имена. Полиморфизмът не е overloading или overriding. Полиморфизмът касае само до приложение със специфично изпълнение(представяне) на интерфейс или по-общо базов клас. Методът overloading се отнася до методи с еднакво име, но различни сигнатути. Методът overriding(динамичен полиморфизъм) означава да се предостави нова реализация на дадения метод, различна от наследената от суперкласа реализация. Новата реализация в подкласа има същото име, същия брой и тип на параметрите и връща същия резултат като реализацията на метода в подкласа.
Пример: Методът Speak() за куче ще върне лае, а за прасе същият ще върне грухти. Кучето и прасето наследяват метода Speak от животно, но методите на подкласа override методи на базовия клас известно като overring полиморфизъм.

32. Виртуални функции. Викане на виртуални функции на базов клас.
Виртуалната функция представлява член-функция, която се дефинира в базовия клас и се предефинира в производния клас. За да се създаде виртуална функция, преди декларацията на функцията трябва да се добави ключовата дума virtual. Когато виртуална функция се предефинира в производен клас, ключовата дума virtual не е необходима.

Чрез виртуалните функции се постига полиморфизъм по време на изпълнение. За целта виртуалната функция трябва да бъде извикана посредством указател. Когато указател към базов клас сочи към обект от производен клас, съдържащ виртуална функция, и тази функция се извика посредством указател, C++ решава коя версия на функцията ще бъде извикана въз основа на типа на обекта, сочен от указателя. А това решение се взема по време на изпълнение.
Пример:

```
int main()
{
  area *p;
  rectangle r;
  p = &r;
  cout << ”Rectangle has area: ” << p->getarea() << ’\n’;
  ...
  return 0;
}
```

33. Вгаждане на обекти. Copy – конструктори.
Copy-конструктор се нарича конструктор, който се вика когато се инициализира даден обект чрез друг:
myclass x=y; //y явно инициализира x
myclass x(y); //същото
И във двата случая се вика copy конструктор. Трябва се отбележи че myclass x=y; не е оператор за присвояване ами инициализиране, т.е. ако се предефинира оператора за присвояване =, няма да влияе в този случаи. Ако не сме дефинирали copy конструктор, компилаторът ни дефинира автоматично за нас.
Copy конструкторът има следният синтаксис:
<име-на-клас> (<име-на-клас> &<обект>)
{ ... }
В него се подава референтно обекта-източник, т.е. самият обект а не негово копие.
Пример:
myclass::myclass (myclass &other)
{ … }
Макар че автоматично се създава copy конструктор е хубаво ние да дефинираме наш си тъй като автоматично създадения е съвсем елементарен и просто копира полетата от обекта-източник в нашия обект. В случаи че имаме указатели сочещи към обекти, то ще се копира само адреса и така ще имаме 2 инстанции имащи за поле указател сочещ на едно и също място в паметта. Това поражда опасност при освобождаване на тази памет. Този частен случаи се избягва чрез copy конструкторът в които ние може да дефинираме как да копира другият обект в нашият, а именно да създаде копие на обектите сочени с указател, и да ги запомниме в нашия обект.

38. Заделяне на обекти от динамичната памет. Проблеми породени от взаимодействията между обекти.
Създадените инстанции от даден клас съществуват докато не излязат извън обсега на действие (scope, края на блока). За да запазиме инстанцията извън scope-а, може да я създадем, като я заделим от динамичната памет където тя ще стои активна докато не я освободим ръчно. Заделянето става чрез new, а изтриването чрез delete.

```
double *buff = new double[1024];
Cat *pCat = new Cat(…);
delete [] buff; delete pCat;
```

Инстанциите биват заделени в паметта, но за да ги достъпим трябва да използваме указател сочещ към тях. Предимството е, че указателя ще изчезне когато излезе от scope-а, но не и инстанцията. Достъпът до полета и методи до дадена инстанция чрез указател става чрез стрелка '->': pCat->speak(); Указателите ни позволяват да ги насочваме към инстанции от клас който е наследен спрямо типа на указателя. Така програмата става гъвкава и съвместима. Пример:

```
Animal *ptr = new Cat(…);
ptr = new Dog(…);
ptr->speak();
```

В този случай ще се извика speak() на класът Animal, освен ако той не е виртуален. Ако е, ще се извика методът на класът Dog наследяващ класа Animal, ако го има. Възможно е и конвертиране на класа от един тип към друг чрез cast-ване, но е опасно.
При работата със указатели, трябва да се внимава да не загубиме указател сочещ към дадена инстанция, без да сме освободили инстанцията предварително, иначе се получава memory leak и те никога в програмата няма да могат да бъдат освободени.

39. Виртуални деструктори

Деструктурът е функция, която се извиква при унищожаването на даден обект. Името на деструктура на един клас е името на класа, предшествано от ~. Локалните обекти се унищожават, когато излязат извън областта си на видимост, а глобалните – когато завърши изпълнението на програмата. Ако е заделен ръчно, той бива освободен при изход на програмата или чрез delete.
При наследяването на класове дестурктурът на един производен клас трябва да се изпълни преди деструктура на базовия клас. Ако деструкторът на базовия клас се изпълняваше пръв, това би довело до унищожаването на производния клас.
Пример:

```
class Base {
   public:
      ~Base(){ ... }
};
class Derived: public Base {
   public:
       ~Derived(){ ... }
};

void main() {
    Base *Var = new Derived();
    delete Var;
}
```

Създава се указател от типа на базовия клас, сочещ към нов обект от типа на производния. При изтриване на указателя от типа на базовия клас директно ще се извика деструктора на базовия клас. Когато се извика деструктора на базовия клас, полетата липсващи в базовия клас Base специфични за наследения клас няма да бъдат унищожени => разрушаването на производния клас ще остане незавършено и специфичните полета няма да бъдат освободени от паметта т.е. програмата ще стане следното:
Constructor: Base
Constructor: Derived
Destructor : Base
Проблемът се решава като се обяви деструкторът на базовия клас за виртуален. Обявяването на деструктора на един клас за виртуален води до това, че всички деструктори на класове по-надолу от него в йерархията се разглеждат като виртуални и всичките се извикват в правилния ред. Т.е. ако сложиме:

```
virtual ~Base(){ ... }
```

В този случай като изтрием указателя от базовия клас, компилаторът ще провери кой подходящ конструктор да извика, а именно този на наследеният.

40.Приятелски класове и приятелски функции.

Приятелската функция не е член-функция на класа, но тя има достъп до *private* членовете на класа. Приятелски функции се използват, когато е необходимо една функция да има достъп до *private* членовете на даден клас.

Декларацията на приятелската функция, в един клас, може да се намира в *public* или в *private* и се характеризира с ключовата дума **friend**, която се поставя в началото на декларацията на метода. Една приятелска функция се дефинира като обикновен метод, но не е член на класът. Затова те не получават като първи параметър по неявен начин **this**. Поради тази причина е необходимо подаването на обект по явен начин като параметър на функцията.

Когато една член-функция прави обръщение към private елемент, компилатора знае че този метод е от този клас и има достъп до това поле. Една приятелска функция обаче не е свързана с обект. На нея просто и е предоставен достъп до private елементите на даден клас.

Приятелска фуннкция не се наследява т.е когато един базов клас съдържа приятелска функция, тя не е приятелска на производния клас.

Един клас за да бъде приятелски на друг клас трябжа да буде деклариран във въпросният клас.Това се получава по подобен начин както приятелските функции, само че пишем директно името на класа след friend.

41. Статични членове на клас.

По подразбиране всички членове са по инстанция(т.е не састатични)

Статичните членове:

- се създават, когато приложението, съдържащо класа се зареди
- съществуват през целия живот на приложението
- имат само едно копие, независимо колко обекта от този клас са създадени
- Достъпват се през класа (НЕ МОГАТ ДА СЕ ДОСТЪПВАТ ПРЕЗ ИНСТАНЦИЯ)

```
class Orbiter {
   ...
 public:
   static int nCount;
};
```

Ако статичният член е деклариран като private трябва да се добави public статична променлива и да се работи с нея.

Статичните данни се използват в конструкторите, когато в run-time се решава какъв обект да се използва.

42. Припокриване на оператори. Същност, ограничения. Реализация.

Предефиниране на операции: В C++ всяка съществуваща унарна или бинарна операция, в която участва поне един обект от някакъв клас, може да бъде предефинирана от програмиста. Това дава възможност класовете да бъдат интерпретирани, като нови типове данни, които могат да се използват по начин аналогичен на базовите типове. Така например, ако бъде дефиниран клас вектор и v1 и v2 са обекти от този клас, то е възможно използването на изрази като: v1+v2, v1-v2, v1*v2, v1/v2...

Предефинирането на операции се осъществява чрез *операторни функции*.
Операторната функция е член-функция или приятелска функция на класа, за който е дефинирана.

Предефиниране чрез член-функции.
Общата форма на една член-функция оператор е следната:
<тип-резултат> <име-на-клас>::<operator>#(<списък аргументи>)
{ ... }
Типът на връщания резултат най-често е същият като типа на класа, за който е дефиниран операторът. Операторът, който се предефинира, замества # в конструкцията.
Предефиниране на бинарни оператори.
Когато една операторна член-функция предефинира бинарен оператор, функцията ще приема само един параметър. Този параметър ще получава обекта, който е от дясната страна на оператора. Обектът от лявата страна е този, който генерира обръщението към операторната функция и точно той се предава неявно на функцията посредством указателя this.
Предефиниране на унарни оператори.
Когато се предефинира даден унарен оператор посредством операторна член-функция, функцията не приема параметри. Тъй като операндът е само един, то той е този, който генерира обръщението към операторната функция.
Предефиниране чрез приятелски функции.
Тъй като една приятелска функция не получава **this** указател, в случай на бинарен оператор това означава, че се предават директно и двата операнда. При унарните оператори се предава единствен операнд.

Не може да се използва приятелска функция, за да се предефинира оператора за присвояване. Оператора за присвояване може да бъде предефиниран само от операторна член-функция.
Всички традиционни оператори могат да бъдат припокривани, включително и new, delete и конвертиращите оператори. Не е възможно да бъдат предефенирани ., .*, ::, ?: операторите.

Правила за припокриване:
- Не може да имаме нов оператор
- Не може да се променя броя на операндите, взимани от оператора.
- Не може да се променят приоритетите или асоциавността на операторите

43. Програмни практики: припокриване на аритметични операции.
Припокриващите се оператори са имплементирани като функции.Дефинират се като ‘operatorx’, където ‘x’ е операторът.
Например за припокриване с оператор за сумиране, можете да дефинирате функцията като operator+.
Подобно е положението и при изваждане и равенство “-“ и “=”.
Операторите, използвани в c++ работят само с примитивни типове, не със наши собствено създадени, тъй като компилатора незнае как да използва операторите за нашите типове.
Всички традиционни оператори могат да бъдат припокривани, включително и new, delete и конвертиращите оператори. Не е възможно да бъдат предефенирани ., .*, ::, ?: операторите.

Правила за припокриване:
- Не може да имаме нов оператор
- Не може да се променя броя на операндите, взимани от оператора.
- Не може да се променят приоритетите или асоциавността на операторите
За пример * винаги има по-висок приоритет от +.

```
struct Complex {
    ....
    Complex operator+( Complex &other );
    };

Complex Complex::operator+( Complex &other )
    { return Complex( re + other.re, im + other.im ); }
```

44. Преобразувания и операции -преобразувания.
Преобразованията е когато сменяме типът на един обект/променлива във друг (още се нарича cast-ване). Има 2 типа преобразования: автоматични и частни, за наще класове. Автоматичните се получават при примитивните типове като int, float, double и т.н.
Пример:

```
int a = 5.3; //a=5;
double b = sin(a); //sin очаква double -> компилаторът го преобразува.
```

Когато обаче създаваме наши класове компилаторът незнае как да преобразува дадения клас в друг тип. Трябва ние сами да опишем метода по който ще става това „преобразуване". Пример:

```
class MyString {
      char *s;
      int size;
    public:
      operator const char * () { return s; } //conversion operator
    ...
};

void main() {
    MyString str("Hello");
    strcmp(str, "Hello");  //Тук се извиква оператора за преобразуване.
}
```

Ако операторът за конверсация липсваше, то когато се извика strcmp, която очаква char*, а му се подаде MyString str, компилаторът ще покаже грешка защото незнае как да преобразува MyString във char*. Слагайки този „метод", той се извиква при нужда от превръщане в този тип. Особеност при този „метод" е че не се задава тип на връщаната стойност (пред оператор), и че винаги няма аргументи.

48. Обектен дизайн : принцип на Лисков.
Функции които ползват указатели/референции към базов клас, трябва да могат да могат да ползват обекти от наследени на този клас, без да знаят за това:

```
Animal *ptr = new Cat(…);
ptr = new Dog(…);
ptr->speak();
```

В този случай ще се извика speak() на класът Animal, освен ако той не е виртуален. Ако е, ще се извика методът на класът Dog наследяващ класа Animal, ако го има.
Това понякога поражда проблеми и за това е препоръчително наследяващия клас само да разширява базовия клас, без да изменя неговата функционалност/поведение.
Пример: square класът наследява rectangle класът като припокрива ВИРТУАЛНИТЕ функции:

```
void square::setWidth(int w) { width = w; height = w; }
void square::setWidth(int w) { width = h; height = h; }
```
След което ако се извиква следната функция ше се види грешка в логиката:
```
void f(rectangle *ptr)
{
  ptr->setWidth(5);
  ptr->setHeight(10);
  ptr->printArea(); //width*height
}
```
Ако към функцията се подаде rectangle инстанция, всичко ще бъде наред и ще излезе 50. Но ако подадем square, ще върне 100, което не е очаквания резултат, гледайки го като rectangle.

49. Поглед към съвременните езикови тенденции и парадигми: обектна ориентация; функционално програмиране.

Обектна Ориентация е от най-известните парадигми, която ни позволява да опишеме обектите във света и техните взаимодействия.

Към ОО се отнасят дефиниране на типове, полиморфизъм, видимост които водят до универсалност и капсулация.

ОО езиците обикновенно са статично типизирани, което означава че всички типове използвани от програмата се проверяват още по време на компилиране. Това предпазва от несъответствие на методи и типове, помагайки за откриване на грешки още преди да се създаде програмата.

Недостатъците са, че това кара програмистът да разчита на това, че в програмата няма грешки и тя ще работи правилно. Също така ограничава свободата при бързото развиване на един проект, разработван от много екипи. При честа промяна на типовете данни с които се работи, клиентът трябва често да прави update-и на своя код/програма.

Функционалното програмиране третира изчислението на дадена задача като математическа функция. За да реши по-голяма задача, то тя се разбива на по-малки парчета код представени като функции, след което комбинира тези късове-функции така че да получи решението на задачата. За разлика от ОО тук не се използват типове на данните и обекти, стойностните са моментни. Действа се на обратния принцип на ОО. Това спомага за по-голяма точност и липса на странични ефекти.

50.Поглед към съвременните езикови тенденции и парадигми: динамични езици; LINQ; декларативно програмиране; логическо програмиране

Парадигмата на програмиране представлява парадигматичен стил на програмиране.

Парадигмата на програмиране предоставя (и определя) начина, по който програмистът гледа на изпълнението на програмата. Например, в обектно-ориентираното програмиране програмистът може да разглежда програмата като съвкупност от взаимодействащи си обекти, докато при функционалното програмиране програмата може да бъде разглеждана като последователност от изчисления на функции, които нямат свои състояния.

Различните програмни езици препоръчват различни парадигми на програмиране. Някои езици са проектирани да поддържат някоя определена парадигма на програмиране - SmallTalk и Java поддържат обектно-ориентираното програмиране, докато Lisp поддържа функционалното програмиране. Някои езици за програмиране поддържат повече от една парадигма - C++ поддържа и обектно-ориентирано програмиране, и процедурно програмиране.

Много парадигми на програмиране са познати повече с това какво забраняват, от колкото с това какво препоръчват. Например функционалното програмиране забранява използването на странични ефекти, структурното програмиране забранява употребата на goto. Заради това новите парадигми обикновено се приемат като твърде сурови и ограничаващи от програмистите, свикнали вече с някакъв предишен стил на програмиране. Но трябва да се отбележи, че избягването на определени техники може да подпомогне доказването на коректността на програмата или просто да улесни разбирането на поведението ѝ, като при това не ограничава приложимостта на програмния език.
Отношението между парадигмите на програмиране и програмните езици може да бъде доста сложно, тъй като един език може да поддържа повече от една парадигма. Например C++ е проектиран да поддържа елементи от процедурното програмиране, обектно-ориентираното програмиране, обектно-базираното програмиране и родовото програмиране. Един C++ програмист може да напише чисто процедурна програма, или чисто обектно-ориентирана програма, а може да напише и програма, която обединява елементи и от двете парадигми.

LINQ is a programming model that introduces queries as a first-class concept into any Microsoft .NET language. However, complete support for LINQ requires some extensions in the language used. These extensions boost productivity, thereby providing a shorter, meaningful, and expressive syntax to manipulate data.

Логическо програмиране

Подход за формализиране обработката на твърдения.

Същност:

Логическа програма = База знания + Целеви твърдения

База знания = Задачи + Правила

Задачи - набор от логически аксиоми (факти)

Правила - правила за извеждане

Целеви твърдения - формулиране на запитванията към базата знания

Изпълнение на логическа програма - доказване на целеви твърдения за конкретна база знания.

Подходи при логическото програмиране

Описателен:

Описание на това как да се направи нещо. Намира приложение в езика LISP.

Декларативен:

Описание на това, какво трябва да се направи. Този подход намира приложение в различни реализации на езика PROLOG.

Програмни езици описващи процесите на изкуствения интелект:

IPL- програмен език за обработка на информация. Той дава възможност на програмата да обработва вместо числа, понятия. Явява се първото средство за имитация функциите на мислене.

LISP - предназначен за описание на задачи за символно представяне и обработка на произволни обекти.

PROLOG - Осигурява обработка на списъци.

51. Управление на памет в динамичен режим – особености, чести грешки

Динамичното управление на паметта се нарича заделянето на памет за целите на дадена програма по време на изпълнение. Дин. заделената памет остава докато не бъде освободена от програмата или от Garbage Collector-ът.Динамичната се различава от статично заделената памет с това че статичната има определено време на живот. Свободното място където се заделя динамичната памет се нарича heap. Използват се алгоритми Best-Fit или First-Fit за попълване на дупките сред заделената памет.

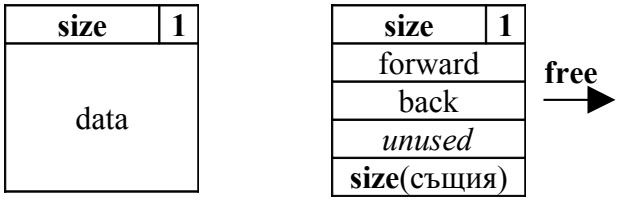
Чести грешки при работа с паметта:
- заделянето на памет- malloc() не занулява/изтрива старите стойности.
- липса на проверка на връщани стойности (returns) и обработка.
- реферииране/указване към освободена памет. Когато се опитаме да четем от нея няма проблеми, но когато пробваме да пишем ще гръмне грешка.
- Освобождаване (free) на памет по няколко пъти.
- Неосвобождаване на памет и загуба на указател към нея – memory leak.
- не съответстващо използване между двойките new/delete и malloc/free()

52. Управление на памет в конзолен режим и в Linux системи: структури в паметта, повреждане на структурите при неправилно менажиране на памет.
В повечето Linux системи за алокиране на паметта се използва принципът на Doug Lea, който е доста бърз и ефикасен. Използва стратегия Best-Fit, т.е. пре-използва освободените (free) парчета (chunks) памет със най-малки загуби. При освобождаване се сращват парчета в по-големи.

Техниката на Doug Lea се базира на двустранно свързани списъци от ОСВОБОДЕНИТЕ парчета.

При този метод, алокираните парчета имат следния формат: **[size]**{data}. **[size]** е размерът на {data}, парчето. Примерно malloc(8) ще задели 8 байта + **[size]** и ще върне адреса на парчето в паметта, където ще пише **[8]**{…}. Ако сме заделили повече парчета памети ще изглеждат така: **[8]**{…} **[32]**{……} **[4]**{..} **[128]**{……}, долепени едно до друго (незадължително). Така ако си на първото парче и искаш да стигнеш до 3-то парче ще скочиш един път 8 байта + още 32 байта надясно и ще стъпиш на 3-то парче. Във **[size]** се включва и допълнителен бит - PREV_INUSE накрая, който посочва дали ПРЕДИШНОТО парче е освободено в момента.

Когато освободиме парче памет, ние само го маркираме че то е освободено. На мястото на старта информация където е било{data} се записва друга служебна информация, а именно forward указател и back указател, както и още веднъж **[size]**. По средата се намира unused секция с неизползваема информация, останала от старите данни, може и да липсва. Структурата на едно освободено парче памет приема следният вид:



Forward и back указателите сочат следващото/предишното свободно парче. Пази се списък от освободените парчета, а не запазените. Когато се опита да се алокира ново парче, започва да се обхожда списъкът докато не се намери подходящо свободно парче, то се откъсва от списъка и върху което да алокира желаната памет. Използвайки това че

са списък, допълнителния бит PREV_INUSE и повтарящия се **[size]** накрая на освободените парчета, може да се направи лесна дефрагментация на паметта, и две съседни свободни парчета да бъдат слепени.
Пример: **[8]**{…} ***[32]****{...free...}* ***[8]****{free}* **[4]**{..} **[8]***{free}* ще бъдат открити че стоят заедно и ще бъдат сляти: **[8]**{…} ***[40]****{...free...}* **[4]**{..} **[8]***{free}*.

53. Препълване на буфер – опасности и използване за недобросъвестно вмъкване на код. Опасности при двойно освобождаване на памет.

Препълване на буфер се нарича, когато процесът пише върху буфер извън заделената за тази програма памет. Така се пренаписва неправомерно съседна памет, където може да се съдържат други променливи, или функции, което може да доведе до грешки при достъп на тази памет, неправилна работа на програмата или пробив в сигурността. Най-често се причинява при липса на проверка за размерността/граници на масива, особено при въвеждане на данни.

Пример за грешки при Doug Lea подредба на паметта:

```
#define BADSTR "…….."
int main() {
    char *first, *second;
    first = malloc(666);
    second = malloc(12);

    free(second);
    strcpy(first, BADSTR); //BADSTR съдържа вреден низ
    return 0;
}
```

Забележка: един char символ е 1Byte! Низът „hello” е от 5 char символи→5 Byte подред.
Когато заделим first който е по голямо парче- 666 байта, очакваме че при заделянето на second ще бъдат заделени непосредствено след first, тъй като е малко парче:
[?]{…}**[666]**{……..}**[12]**{…} **[?]**{…}
Когато се извика **free**(second); това парче бива „освободено” и вкарано в свързаният списък с свободни парчета. В самото парче старите данни {data} биват пренаписани със служебна информация като forward pointer и back pointer.
Така подредени, се изпълнява strcpy(first, BADSTR), която взима низът от втория аргумент и го копира на първия, без никакви проверки. В такъв случаи ако сме заделили 666 байта за first, а подадем низ BADSTR с размер 668, то ще бъдат пренаписани 668 байта от началото на first надясно т.е. 2 байта ще бъдат пренаписани върху следващата клетка, а именно **[12]** полето ще бъде засегнато. Ако запишеме повече байтове и {...} полето ще бъде засегнато. Тъй като second парчето е вече освободено, в него пише служебна информация- указатели. Така може да пренапишем {…} полето така че да пренасочим указателите към други адреси.

По-подобни начини може да бъде изменена/повредена структората на паметта, да бъдат пренасочени собствени функции към функциии изпълняващи зловреден код или най-малкото да променим действието на програмата. Такива атаки може да използват **unlink**, **frontlink**, двойно освобождаване на паметта и т.н. Дефрагментацията и слепването също спомагат за това.
Повторно освобождаване на вече освободена памет води до объркване на логиката на списъците, което също е опасно.