**1. Структури и обединения. Представяния в паметта. Размер на обекти/истанции. Подравняване. Подаване на обекти/истанции във функции. Създаване на обекти в динамичната памет. Примери.**

* Struct

1. User-defined data type
2. Групираме елементи, наричани членове, от различни типове в едно (пример)
3. Паметта, която заема, се изчислява на база членовете му + подравняването
4. By default всички членове са public
5. Елементите се достъпват с точка
6. Инициализация, може да имаш конструктор (пример)

* Union

1. Тип структура, групира данни и т.н
2. Като синтаксис наследява всичко от структурите
3. Променливите в един union заемат едно и също място в паметта, с размер равен на най-голямата променлива (при промяна на една променяш всички)
4. Пример a=75

Diagram

Description automatically generated

* Представяния в паметта. Размер на обекти. Подравняване

1. Байтовете на една променлива се записват в обратна посока. Най-старшият байт е последен.
2. Всеки байт в паметта може да се достъпи индивидуално
3. В паметта променливите са разположени в същия ред, в който са декларирани в стурктурата
4. Int – 4 byte-a естествено подравняване, short – 2, char – 1
5. Подравняването се отнася към начина, по който променливите биват разположени в клетките в паметта, става по големината на най-голямата променлива

Text

Description automatically generated

В случая най-голямата int – 4 byte => подравняваме на 4 byte, short – 2byte + char 1byte => 1 byte padding, за да допълним до 4, след това поставяме int => общо 8 byte-a памет, ако int-a беше между short и char, към short(2byte) – 2byte padding, за да допълним до 4, след това int-a, след това char(1byte) с 3byte-a padding => общо 12 byte-a памет

1. 32bit архитектура – чете 4 байта при един цикъл – не можеш да прочетеш double на един път, 64 => 8 байта на цикъл

* Подаване на обекти към функции

1. Const Object& (Подаваме по константна референция, във функцията, ще работим със същия обект, който сме подали, няма да правим негово копие, но няма да можем да го променяме в scope-a на функцията, могат да се извикват само константни функции на обекта)
2. Object& (Подаваме по референция, във функцията, ще работим със същия обект, който сме подали, няма да правим негово копие и ще можем да променяме неговите член-данни)
3. Object (Подаваме по копие, във функцията се създава копие на подадения обект, чрез copy constructor)

* Създаване на обекти в динамичната памет
  1. Създаването на обект в динамичната памет става с operator new, връща указател към заделената памет в heap-a (int \* a = new int(4))

Бонус: Паметта в C++ се разделя най-общо на 3 вида

1. Статична – статични/глобални променливи
2. Stack – локални променливи, дефинирани в телата на функициите
   * + Заделя се в момента на дефиниция на променливата
     + Освобождава се в момента на изход от блока, в който е дефинирана
     + Последно заделената се освобождава първа! (Stack е все пак)
     + Нямаме контрол – не можем да освобождаваме, когато пожелаем
     + Количеството заделена памет се определя по време на компилация
3. Heap – използва се динамично в хода на програмата
   * + Може да бъде заделена и освободена по всяко време
     + Областта на динамичната памет е набор от свободни блокове памет
     + Можем да заявим блок с произволна големина
     + OS се грижи за управлението на динамичната памет

2. **Текстови файлове. Потоци за вход/изход от файл. Режими на работа. Флагове на състоянията на потока. Пример: програма, която отпечатва собствения си код. Пример: функция, която връща големината на файл.**

* Поток – последователност от байтове данни, влизащи в или излизащи от програмата. Input/Output операциите са базирани на потоци. Служат като посредници между програмите и IO устройствата.
* Cin, cout – потоци за вход и изход от конзола, операто >>, <<
* Fstream – от библиотеката fstream, поток за вход/изход – приема име на файл и режим на работа, f.open(), f.is\_open(), f.close(), ifstream и ofstream
* Четене/писане, >>/<<, get, getline
* Състояния на поток

1. bad() – връща true, ако е възникнала грешка при четене/писане. Ако потокът е bad, то не могат да се извършват повече операции върху него
2. fail() – връща true, ако е възникнала грешка, но все пак още могат да се извършват операции върху него
3. eof() – връща true, ако е достигнат края на файла
4. good() – връща true, ако потокът не е в нито едно от трите предишни състояниа

clear() – изчиства всички състояния и слага потока в състояние good

* При четене от и писане във файл, данните са посочени от специален указател

1. tellg() – връща позицията на текущия символ в потока за четене
2. tellp() – връща позицията на тeкущия символ в потока за писане
3. seekg(offset, direction) – премества get-указателя с offset на брой позиции, започвайки от позицията direction на позоцията на потока
4. seekp(offset, direction) - премества put-указателя с offset на брой позиции, започвайки от позицията direction на позоцията на потока
5. seekp(streampos idx), seekg(streampos idx) – премества put-указателя на позиция idx

offset – целочислена стойност, отместване от direction

direction – ios::beg, ios::cur, ios::end

* Режими на работа (a|b|c) побитово “или” със степени на двойката

1. Ios::in – отваря файла в режим за четене ifstream
2. Ios::out – отваря файла в режим за писане ofstream
3. Ios::ate – отваря за вмъкване и установява указателя put в края на файла, допуска вмъкване на произволни места
4. Ios::app – отвяра за вмъкване и установява указателя put в края на файла, всички вмъквания са в края на файла
5. Ios::trunc – ако файлът съществува, съдържанието се изтрива
6. Ios::binary – превключва режима от текстов на двоичен
7. Ios::nocreate – отваря за вмъкване, само ако файлът с указаното име съществува
8. Ios::noreplace – отваря файлът за вмъкване само ако файлът с указаното име не съществува

* Human-readable - .txt format, удобни за човека, но не и за програмата
* <https://github.com/Justsvetoslavov/Object-oriented_programming_FMI_2021-2022/blob/main/Sem.%2002/task1.cpp>

3. **Двоични файлове. Режими на работа. Флагове на състоянията на потока. Позициониране във файл. Запазване на обекти в двоичен файл. Пример: запазване/четене на масив от обекти(от един тип) във файл.**

* Потокът се отваря в режим на работа ios::binary
* Computer-readable – удобно за компютъра, но не и за човека
* Четене – read(char\* memoryBlock, size\_t size) – винаги преобразуване указателя в char\*
* Писане – write(const char\* memoryBlock, size\_t size)
* Записва се и padding-a
* Съдържание – байтовете се записват в обратна посока, най-старшият байт е последен
* Graphical user interface, text, application, chat or text message

  Description automatically generated - 155555
* <https://github.com/Justsvetoslavov/Object-oriented_programming_FMI_2021-2022/tree/main/Sem.%2003>

4. **Член-функции. Mодификатори за достъп. Капсулация. Конструктори и деструктор. Извикване на конструктори и деструктори. Конструктори и деструктор при композиция на обекти. Примери.**

* Член-функции

1. Функция дефинирана в scope-a на класа
2. Компилаторът първо компилира декларациите, след това телата на член-функциите, затово дефинициите могат да бъдат извън класа
3. Имат достъп до компонентите на класа, като всяка функция има this pointer, който сочи към конкретната инстанция на класа и е константен поинтър(не можем да променим към какво сочи)
4. Компилаторът превежда член-функциите в обикновени функции с уникално име, които винаги получават поинтър към обекта(this) като първи параметър
5. Константни член-функции не правят промени върху обекта, върху който са извикани в scope-a на функцията, индикират се с const суфикс след параметрите, могат да се извикват от const обекти

* Модификатори за достъп. Капсулация
  1. Капсулация – скриване на информацията, основен ООП принцип
  2. Този принцип позволява да определим кои методи и атрибути могат да бъдат достъпвани от външния свят
  3. Private – можем да ги достъпим само в текущия клас
  4. Protected – в текущия клас и наследниците
  5. Public – всеки
  6. Getter - достъп и setter - промяна
* Конструктори и деструктори и тяхното извикване

1. Конструктор
   * + Извиква се веднъж при създаване на обекта, за да инициализира данните на обекта
     + не се оказва експлицитно тип на връщане(връща const ref)
     + Със същото име като класа
     + Може да има параметри
     + Initializer list – задава начални стойности на данните, изпълнява се преди да влезе в body-то на constructor-a
     + При липсата на експлицитно създаден, компилаторът експлицитно създава default, който рекурсивно default-ните на член данни(case те да нямат default-ен!)
2. Деструктор
   * + Извиква се веднъж при изтриване на обекта
     + Не се оказва експлицитно тип на връщане
     + Има същото име като класа с ~ отпред
     + Вика се, когато обекта излезе от scope-a, където е деклариран
     + Когато е извикан delete[]/delete върху динамично заделен обект/и от съответния тип
3. При липса на конструктори и деструктори - компилаторът дефинира автоматично такива по подразбиране
4. Ние ги дефинираме, понеже тези по подразбиране не правят винаги това, което бихме искали – работа с динамична памет

* Конструктори и деструктори при композиция

1. Композиция – класове като елементи на други класове
2. В инициализер list се оказва, кои конструктори на член-данните да се извикат, ако не се уточни => default-ни
3. Деструкторите на член-данните се извикват в обратен ред на това как са подредени като член-данни
4. Деструкторите на член-данните не трябва да се извикват експлицитно в деструктора на класа
5. Първо се вика деструктора на класа, в който е композицията, след това деструкторите на класовете от композицията

5. **Разделна компилация. Абстракция. Примери.**

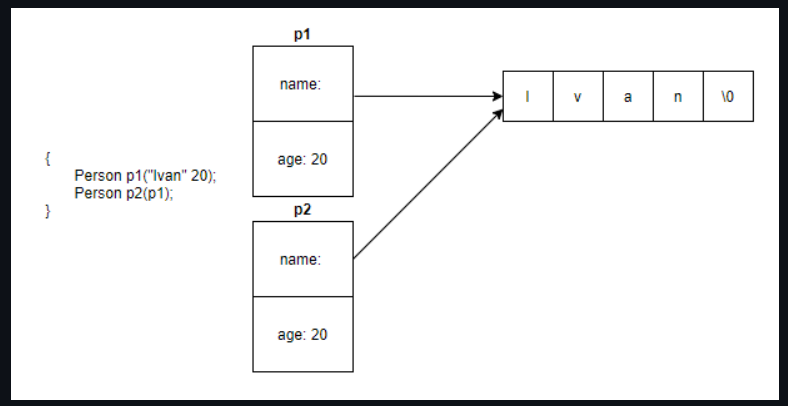
* Разделна компилация
  1. Защо е нужно разбиването на файлове? (Малки промени – прекомпилира се всичко, усложнява се съвместната работа, усложнява се преизползването)
  2. Проектът се разделя на множество изходни(.cpp) файлове, като всеки се компилира независимо от другите
  3. Първоначално, файлът се дава на препроцесора, който изпълнява всички директиви (#include etc.). Първоначално при компилация, изходният файл се предава на препроцесора, който заменя #include със съответното парче код, което се съдържа във файла. След това изходните файлове превръщат в обектни (.obj) файлове с машинен код. Минава linker, който свързва всички .obj файлове в изпълним код (.exe). Той асоциира всички референции към имена на променливи, функции и т.н. на един обектов файл към съответните им дефиниции, които могат да се намират в други изходни файлове. Ако не се намери дефиницията в другите файлове или в стандартните C++ и C библиотеки, linker-a хвърля грешка.
  4. Компилаторът хвърля грешка, ако не намери декларация, linker-а хвърля грешка, ако не намери дефиниция
  5. .cpp файлове – съдържат дефиниции, компилират се
  6. .h файлове - съдържат декларациите на функциите, до които искаме да има достъп външния свят, не се компилират, а се вмъкват във файлове, които се компилират
  7. .hpp – pure c++ files
  8. #pragma once – директива, която казва на компилатора да обработи този код само веднъж
  9. Header guard-ове – уникален идентификатор, #ifndef LIBRARY\_H #define LIBRARY\_H #endif-a
* Абстракция

1. Скрива цялата логика, която не ни интересува или не ни е нужна, виждаме само резултата от нея, който ни трябва/информацията, която ни трябва
2. Пример – триъгълника, с точките

6. **Динамична памет в класовете. Голямата четворка. Пример: клас студент с име (с произволна дължина) и масив от оценки(с произволна дъжина).**

* Голяма четворка

1. Конструктор по подразбиране(конструктор без параметри)
2. Copy constructor
3. Copy assignment operator
4. Destructor

* By default компилаторът създава голямата четворка, като всяка от тези функции извиква рекурсивно същите функции върху член-данните
* Проблемът се появява при използването на динамична памет в класовете – при copy constructor, copy assignment operator – ще се получат два обекта използващи една динамична памет, в деструктор тя няма да бъде изтрита
* => трябва да се имплементират експлицитно copy constructor, copy assignment operator и destructor
* <https://github.com/Justsvetoslavov/Object-oriented_programming_FMI_2021-2022/tree/main/Sem.%2006/Student>

7. **Предефиниране на оператори. Приятелски класове и функции. Пример: реализация на комплексно число.**

* Оператори

1. Унарни и бинарни (и един тернарен “bool ? true : false”)
2. Имат приоритет – при израз с повече от един оператор, определят последователността на извършване
3. Имат асоциативност – при изрази с оператори от еднакъв порядък, определят последователността
   * + Дясноасоциативен – първо се изпълнява най-десния
     + Лявоасоциатевин – първо се изпълнява най-левия
4. Имат позиция спрямо аргумента – префиксен, инфиксен, суфиксен

* Предефиниране на оператори

1. Позволява да дефинираме поведение на оператор
2. Специални функции с имена operator последван от символа
3. Може да се предефинира като член-функции или като friend-функция
4. Имат толкова на брой параметри, колкото операнди
5. Не могат да се променят приоритета, асоциативността, броят и позицията на аргументите му
6. Дефинираме като член-функции, когато са присвояващи(+=,-=), индексни([]), member access(->), променят състоянието на техния обект(++, --) или достъпват нещо от него
7. Извън класа/friend – симетрични оператори (a X b == b X a)

* Приятелски класове и функции

1. Декларират се запазената дума friend пред прототипа на функцията, единствено в декларацията се пише friend
2. Friend клас/функция има достъп до всички компоненти на класа, независимо дали са private, protected, public

* <https://github.com/Justsvetoslavov/Object-oriented_programming_FMI_2021-2022/tree/main/Sem.%2007/Practicum/Complex>

8. **Move семантики - ползи, lvalue, rvalue, move конструктор/move assignment operator(operator=), std::move. Пример: клас String.**

* Ползи – да спестим излишни копирания – пестим време и памет
* Lvalue – изрази, които притежават някакъв адрес в паметта, например променливи, обекти, функции, връщащи ref към някакъв тип и т.н
* Rvаlue – изрази, които не са lvalue
* Rvalue референция - &&, с тях можем да разпознаваме дали заемат стойност в паметта или не
* Std::move – прави lvalue на Rvalue, казваш, че този обект няма да се ползва повече, force-ваш да се ползват move семантики
* Move operator-и

1. Използваме ги, за да местим ресурси от временни обекти, вместо да правим копия
2. Приемат rvalue референция, която не е const
3. Компилаторът сам решева дали да използва move семантики, но това е възможно само, ако са дефинирани такива

Text

Description automatically generated

9. **Шаблони. Пример: шаблонен клас stack.**

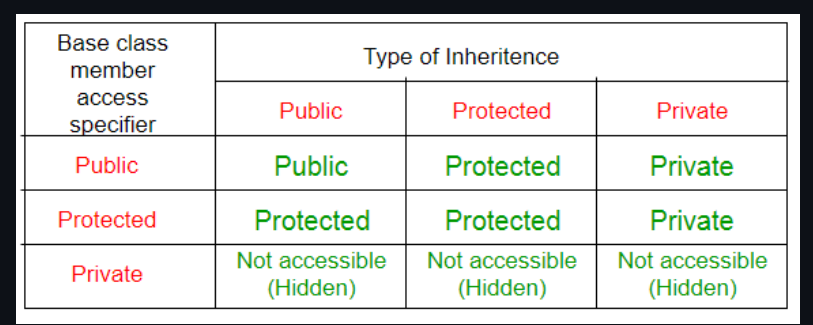
* Функиция/клас, която не работи с променливи от някакъв дeфиниран тип, а с абстрактни променливи, се нарича шаблонна функция
* template<typename T>
* Може да се специализира за конкретен тип да има друга имплементация
* В случая T трябва да изпълнява определени constraints, пример – шаблона ползва operator+= на класа T, но подаваме клас/тип, който няма дефиниран operator+= => не може да се инстанцира, грешката се хваща compile time
* Разписват се в .hpp файлове
* Компилаторът не генерира код от дефиницията на шаблона. Той генерира код едва когата срещне конкретна инстанция на шаблона
* Шаблоните на класове не са конкретни класове, а единствено предоставят спецификация, по която да се създаде един такъв
* <https://github.com/Justsvetoslavov/Object-oriented_programming_FMI_2021-2022/blob/main/Sem.%2010/Stack.h>

10. **Наследяване. Видове наследяване. Конструктури и деструктори при наследяване. Копиране при наследяване.**

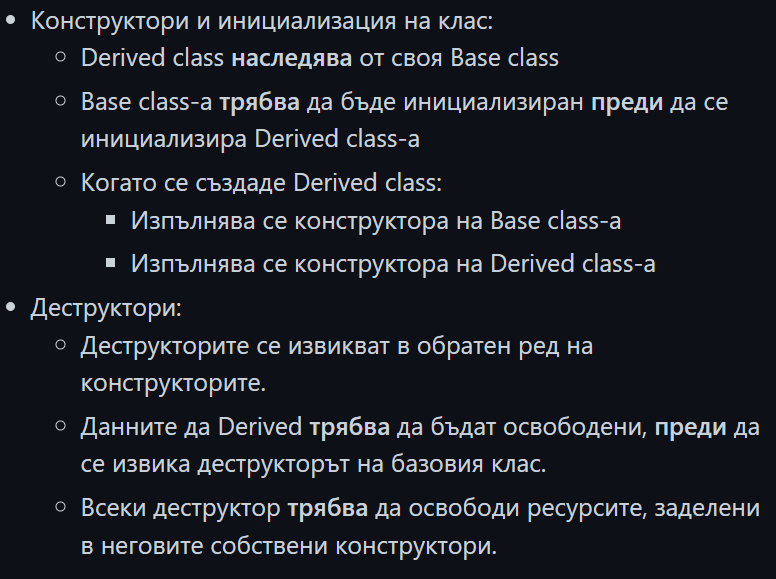
* Наследяване – метод за създаване на нови класове от съществуващи класове, новият клас съдържа данните и поведението на съществуващия клас
  1. Наследяват се член-данни и методи
  2. Получава се достъп до някои от компонентите на основните класове
  3. Is-a relationship vs has-aA screenshot of a computer screen

     Description automatically generated with medium confidence
  4. Наследниците могат да бъдат подавани като параметри на функции, които приемат обекти от базовия клас
* Видове

1. Единично – нов клас се създава от друг единичен клас
2. Множествено – нов клас се създава от два или повече други класа
3. Private –
4. Protected –
5. Public –



* Конструктори и деструктори

1. 

* Копиране при наследяване – при разписване на copy constructor и copy assignment оператор, трябва експлицитно да извикване копиране на базовия клас – пример

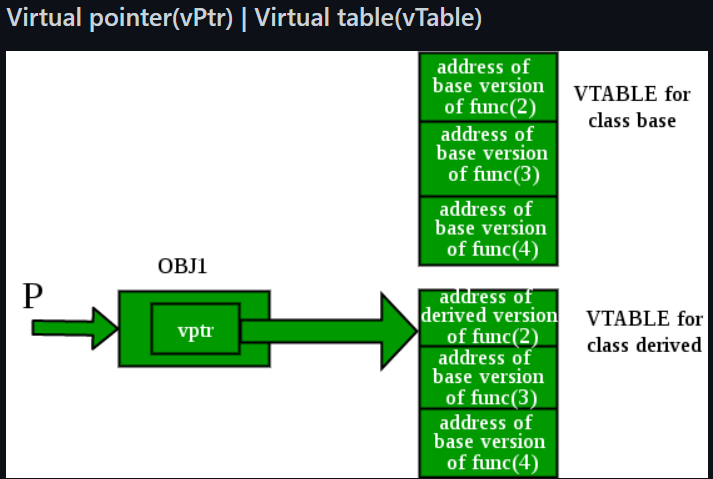
(ако не се извика explicit-но се, се извиква default constructor)

11. **Статично и динамично свързване. Виртуални функции. Виртуални таблици. Ключови думи - override, final.**

* Статично свързване (static binding/early binding) – изборът на функцията, която трябва да се изпълни става compile time, извиква се дефиницията от класа, с който извикваш функцията дори да е pointer/ref към наследник
* Динамично свързване (dynamic binding/late binding) – изборът на функцията, която трябва да се изпълни става по време runtime, постига се с виртуални функции, виртуалната функция намира най-последната дефиниция, независимо от класа, чрез който е извикана
* Виртуални функции

1. Член-функция, която е декларирана в основен клас и е предефинирана от друг клас
2. Гарантира, че правилната функция ще се извика за даден обект, независимо от типа reference/pointer, използван за извикване на функцията
3. Използват се за постигане на runtime polymorphism – това се получава, ползвайки reference/pointer към базовия клас, за да достъпим виртуалната функция
4. Не могат да бъдат static
5. Прототипът трябва да бъде един и същ в базовия и в наследника
6. Дефинират се в базовия клас, предефинират се в наследниците. Ако не бъдат предефинирани се използва имплементацията на базовия клас.
7. Може да има виртуален деструктор, но не може да има виртуален конструктор

* Vtable – Ако в един клас има поне една виртуална функция, за този клас и всеки негов наследник се създава виртуална таблица. Vtable е статична таблица, която държи правилните адреси на виртуалните функциите на даден клас, които трябва да се извикат. При инициализиране на обект той се създава със скрит vptr, който сочи към таблицата на класа, от който е инстанциран. При извикване на виртуална функция се взима vptr, достига се таблицата и се вика правилната функ.



* Override – указва, че дадена функция презаписва функция от базовия клас. Ако в базовия клас няма такава функция, то кодът няма да се компилира. Не е задължителна за писане, но може да спести неочаквани грешки при промяна на кода.
* Final – указва, че дадена ВИРТУАЛНА функция не може да се презаписва надолу по йерархията или че даден клас не може да се наследява

12. **Полиморфизъм. Абстрактни класове.**

* Compile time polymorphism – дадена функция се извиква по време на компилация на програмата(function overload/operator overload)
* Runtime polymorphism – функциите се извикват в момента на изпълнение на програмата (Dynamic binding/Late binding)
* При полиморфна йерархия трием обектите чрез указател от базовия клас. За да се извика привилните деструктори, деструкторът на базовия клас трябва да бъде virtual
* Полиморфизъм – едни и същи действия се реализират по различен начин, в зависимост от обектите, върху които се прилагат

1. Действията се наричат полиморфни
2. Реализират се чрез виртуални функции
3. Класовете, върху които се прилага, трябва да имат общ родител или прародител, т.е да са наследници на един и същ клас
4. В класа се дефинира виртуален метод, съответстващ на полиморфното действие
5. Всеки клас предефинира или не виртуалния метод
6. Активирането става чрез указател към базов клас, на който може да се присвоят адресите на обекти, на който и да е от наследниците

* Абстрактни класове

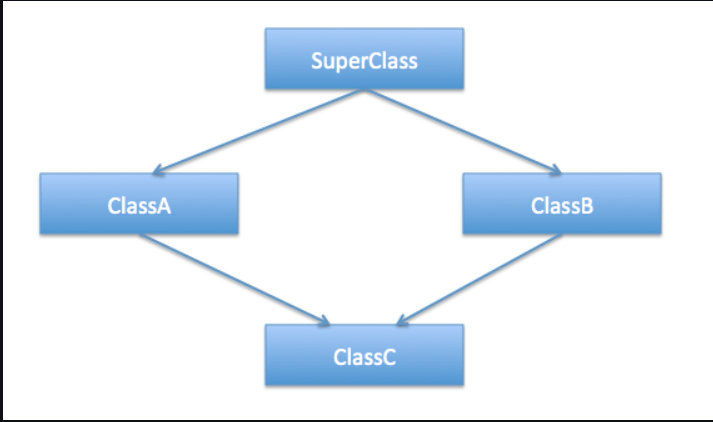
1. Клас, от който не мога да се създават обекти(инстанции), предназначен само за наследяване
2. За да бъде абстрактен – поне една pure virtual функция(виртуална функция без тяло)
3. Ако наследник не разпише собствена имплементация на наследените чисто виртуални функции, става абстрактен

13. **Множествено наследяване. Диамантен проблем. Виртуално наследяване. Пример: полиморфна йерархия с клас Shape.**

* Множествено наследяване

1. Един клас може да наследява, повече от един клас
2. Конструкторите на наследените класове се извикват в ред на наследяване
3. Copy constructor-и и operator= извикват тези на наследените класове

* Диамантен проблем. Виртуално наследяване

1. Получава се при множествено наследяване, когато един клас наследява класове, които имат общ базов клас
2. В случая ще се извика 2 или повече пъти конструктора на базовия клас и на по-късен етап ще се извика два пъти деструктор – проблем
3. Решение ClassA и ClassB да наследяват виртуално SuperClass 
4. След добавяне на virtual преди да се извика конструктор на родител на virtual наследник, всеки път се проверява дали вече е извикна този конструктор
5. Деструкторите се извикват в обратен ред на конструкторите, като там се прави същата проверка дали деструктора вече е извикан
6. Когато използваме keyword: virtual, конструкторът по подразбиране на прародителя се извиква по подразбиране, дори ако родителските класове извикват изрично конструктор с параметри, ако искаме друг ще трябва да викнем експлицитно друг конструктор в класа наследник на виртуалните(ClassC)

* <https://github.com/Justsvetoslavov/Object-oriented_programming_FMI_2021-2022/tree/main/Sem.%2013/ShapeCollection>

14. **Колекции от обекти в полиморфна йерархия(хетерогененконтейнер). Копиране и триене. Пример: клас Farm.**

* Колекция от различни типове с общ базов клас – масив от указатели към базовия клас
* Clone function – виртулна функция декларирана в базовия клас, която се предефинира във всеки от наследниците и връща pointer от BaseClass към копие на самия обект, за да не се наруши абстракцията
* Virtual Destructor в базов клас – за да сме сигурни, че ще се извика първо деструктора на наследника и после всички деструктори нагоре по веригата, докато стигнем базов клас
* Как работи free
* Как работи copy
* <https://github.com/Justsvetoslavov/Object-oriented_programming_FMI_2021-2022/tree/main/Sem.%2013/Farm>

15. **Type casting. SOLID principles.**

* Type casting

1. static\_cast

* Default cast за implicit conversions
* Използва се по default за примитивни типове
* Може да е implicit или explicit
* Compile time
* Прави строга проверка на типа
* Примера с private наследяване
* Text

  Description automatically generated
* Пример с конвертиращ конструктор

1. dynamic\_cast

* Използва се при полиформизъм, нужна е **поне** една виртуална функция
* Downcasting, Upcasting
* Runtime
* При невалиден cast връща nullptr
* A screenshot of a computer

  Description automatically generated with medium confidence - nullptr

1. const\_cast

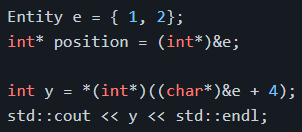
* Използва се при добавяне/премахване на const към променлива
* Добавяне/премахване на constness на променлива
* Пример: да ползваш this в const функция, за да променяшText

  Description automatically generated

1. reinterpreted\_cast – прави type punning

* Използва се за преобразуване на pointer от даден тип към pointer от друг тип, дори когато типовете несъвпадат – не проверява дали двата типа са еднакви => може да доведе до runtime грешки
* Използва се, когато искаме да работим с битове
* A screenshot of a computer

  Description automatically generated with medium confidence

1. Type punning – интерпретиране на един тип данни, като друг, когато двата тима не са свързани по никакъв начин, обикновено с поинтъри 
2. C style cast – със скобички отпред, може да гръмне runtime, прави последователно горните, без const, ако всички гръмнат гърми програмата, изисква повече ресурс заради многото проверки
3. <https://github.com/Justsvetoslavov/Object-oriented_programming_FMI_2021-2022/tree/main/Sem.%2014/Type%20casting>

* SOLID - Пет принципа на проектиране, имащи за цел да направят софтуерния дизайн по-разбираем, гъвкав и лесен за поддържане.

1. Single Responsibly
   * + Всеки модул(например клас), носи отговорност САМО за една цел – един модул върши точно едно нещо
     + Класът има само една причина да се променя – промяна в спецификацията
     + Позволява да се пише софтуер и висока кохезия и стабилност, който не се влияе толкова от развитието на системата
2. Open-Closed Principle
   * + Един модул е отворен за разширение, но затворен за модификация
     + Позволява добавянето на нова модификация без промяна на съществуващ изходен код
     + Позволява да се пише софтуер за многократна употреба и подръжка
     + Може да се постигне чрез полиморфизъм и абстрактни класове
3. Liskov Substitution
   * + Подкласовете трябва да могат да заместят поведенчески своите родители, всеки родител можем да го заменим с child class, без значение кой точно
4. Interface Segregation
   * + Класовете/Interface-ите трябва да бъдат с точно и ясно предназначение
     + Избягваме класове/interfaces, които правят всичко => разцепваме ги логически
     + По-добре много интерфейси с едно предназначение, отколкото един с много.
     + Позволява преизползване и персонализиране, тъй като не е нужно да имаме функционалност, която няма да използваме
5. Dependency Inversion
   * + Нашите класове трябва да зависят от interfaces/abstract-ни класове, а не на на конкретни класове и функции

16. **Design patterns. Singleton & Adapter.**

* Стандартни решения на често срещани проблеми в софтуерния дизайн. Всеки design pattern е шаблон, по който да решиш даден проблем.
* Увеличавант гъвкавостта и преизползването на код, улесняват development-a при скалиране на приложението.
* 3 вида патерни
* Behavioral – улесняват комункацията между обекти
* Observer – предоставя механизъм за закачане на обекти към обект, така че да получнат информация за event-и, които се случват на обекта (notify за промяна, изтриване и т.н.)
* Command – използва обект за да капсулира цялата логика за изпълнение на дадено действие
* Creational – предоставят различни начини за създаване на обекти
* Factory – създава интерфейс или абстрактен клас за създаване на обект, но позволява на наследниците да изберат кой клас да инстанцират
* Abstract Factory – is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.
* Singleton – има само една инстанция на обекта за целия lifetime на програмата, която е достъпна от цялата програма (irl пример – Една държава, едно правителство)
  + 1. Плюсове – достъпва се от всякъде, и има точно една инстанция, която се създава при първо извикване
    2. Минуси – Нарушава S от Solid (хем дава инстанция, хем създава, хем се грижи за lifycycle му), при многонишкова програма, обекта може да бъде създаден няколко пъти(първата влязла нишка трябва да lock-не scope-a на създаване, другите нишки ще изчакват, първата ще създаде обекта и ще release-не lock, след като влязат другите те вече ще видят, че има създаден обект и няма да тръгнат да правят нов), проблеми при unit test-ове, понеже не може да се моква static и конструктора на сингълтън class-a е private
* Structural – отнасят се към това как класове и обекти са композирани, за да образуват по-голяма структура. Улесняват структурата като задават връзка между самите класове
* Adapter – позволя несъвместими интерфейси да работят заедно. Parse-ва данните от едно състояние в друго, което ни е нужно (irl пример – адаптер, че от usb към нещо друго, програма връща json, но друга иска xml => json to xml parser(adapter))
  1. Плюсове – Single responsibity – отделяш парсването между двата интерфейса от бизнес логиката и адаптерът се грижи само за нея, open/closed – можеш лесно да добавяш нови адаптери без да счупиш кода
  2. Минуси – увеличава се сложността на кода – трябва да се добавят нови интерфейси и класове
* Facade – предоставя опростен интерфейс към библиотекта или комплексен набор от класове
* <https://github.com/Justsvetoslavov/Object-oriented_programming_FMI_2021-2022/tree/main/Sem.%2015/Creational%20Patterns/Singleton>
* <https://github.com/Justsvetoslavov/Object-oriented_programming_FMI_2021-2022/tree/main/Sem.%2015/Structural%20Patterns/Adapter>

OOP – Програмна парадигма(начин/стил на програмиране) базирана на класове и обекти. Четири основни принципа – Наследяване, Полиморфизъм, Капсулация, Абстрация

Technical debt – да приоритизираш нещо да стане бързо, но не с перфектен код => в някакъв момент в бъдещето ще трябва да рефакторираш и да го направиш по правилния начин

Text

Description automatically generated