

Структури и обединения. Представяне в паметта
Подразбиране. Раздели на обекти. Предаване
във функции. Динамично заделяне на обекти.

- Структурите/класовете се използват за групирание на
данини (примитивни или структури), имат десетки разлики.

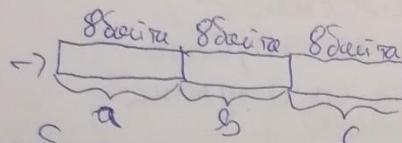
- Създава се нов тип данни.

- В структурите по default е public,
в класове - private (вкл. наследяване).

struct Box {

double a;
double b;
double c;

};

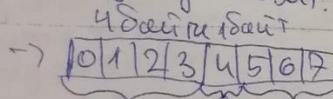


→ общата големина на Box е 24 байта.
→ при подразбиране се записват последният
ред на рег (reg no reg), и в обратен
ред се унищожават.

struct X {

int a;
bool b;

};



→ записването на променливите в паметта започва
от по-дългия с по-реден номер краен на големината и се
продължава.

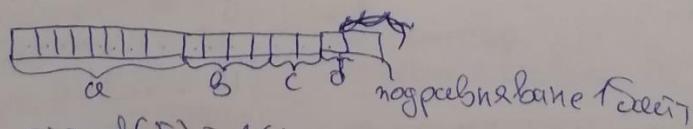
→ Краят на големина е означена за ~~номер~~ и големината на
най-дългата примитивна тип данни. (което не е единично
за обект, а следва иерархията примитивни тип данни).

→ За идей-оптимално разпределение → подразбиране в
към края рег.

struct F {

long long int a;
int b;
short c;
bool d;

};



sizeof(F) = 16;

struct f {
bool d;
char short c;
int b;
long long int a;

};

struct A {

};

→ въвежда дефиниции на
данни. Може и
структурата да се
дефинира в друго, но
тя може да се използва
само чрез имена и
възможности (A::B)

Struct

- данните се записват в на място една след друга в отделни блокове памет!

Union

(обединение)

→ данните споделят една памет.

Union X {

int a;
bool b;
short c;

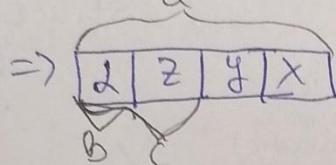
};

→ int a' иди вада:

X	1	9	1	2
---	---	---	---	---

в памети:

2	1	2	1	Y	X
---	---	---	---	---	---



- може ли да адресираме и работим само с една ~~ст~~ гел данните в даден момент, като всички си ~~запамти~~ колкото беше са и нулси.

Градежане на обекти и масиви и издаване на функции.

Box b; // в статичната памет

Box b2 = {2, 3.5, 5};

Box b3;
b3.a = 5;
b3.b = 6;
b3.c = 8.5;

stack →

Box Boxes[10]; // статичен масив с 10 ячии.

Box* ptr; // поинтор в статичната памет

ptr = new Box(); // динамично зареден обект

Box* ptr2 = new Box[10]; // динамичен масив.

(*ptr).a = 5; //

ptr.b = 3.5; // чрез поинтор се използва →

ptr->c = 5.5; //

delete ptr; !!!

delete[] ptr2; !!!

Тип масивите са нулси

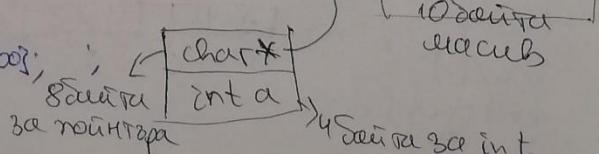
default constructors.

Динамичните масиви са copy конфри:

struct Box {
char* name;
int a;

Box B obj;

obj.name = new char[100];



за поинтара

4 байта за int

Градежник определен по 3 горни

void F(const Box& B)

→ константна референция
за да не копирае обекти и
ако има да го променяме

void F(Box& B)

→ тук може да го променяме

void F(Box B)

→ тук обектът се създава и
ще се извърши ~~из~~ след края на
функцията.

Във функции по-добре
да се издава референция,
отделното поинтор, за да
се избегне проверка за
nullptr.

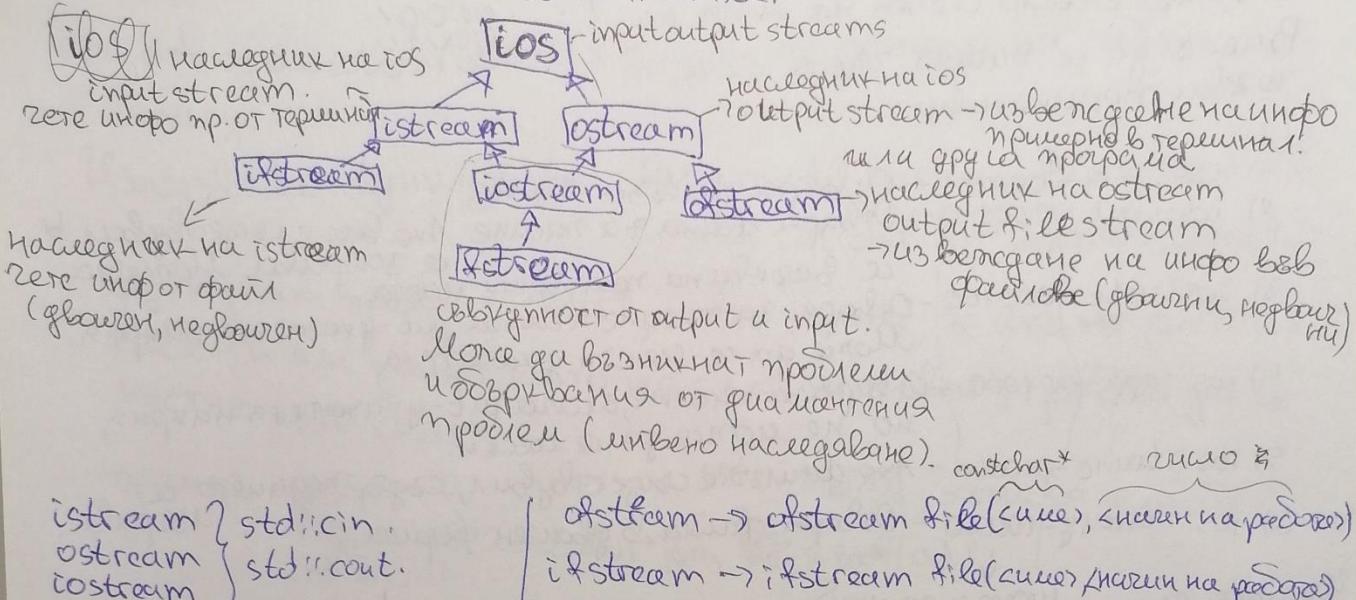
heap

Община
масив

II

Текстовые фреймы. Иерархия и потоки.
Потоки за вход/выход. Ресурсы работы.
Файлы на свидетельстве.

2



След откриване на файлът навсякъде съдържащо проверка файл
се съхранява! if(!file.is_open()){

3

Наредка се срета с ошибка: file.close(); → редко запада на
file.close() → може да западе уникат → где е горе в try catch.

`ifstream`:

```
int a, charc, charline[100];  
file>>a>>c;
```

Предоставиран ~~от~~ от оператор. → разходи
За е със ~~се~~ прозете и баланс за коректно
За е със прозете 1 бал. за син.

За се употреби 1 бант за символ
file.get() → промяна на символа
и натича се get-запис на първия
бранг символ.

file.readline(line, 1024), → go koda
mash kohkoga nu pega
npozete. nu koda.

- get nowtop. → corn xbl all aleg -
бакун суббота, хонго ле пророк
Монгола илчли.
- ed → суббота. бодхара балг,
и цэе энэдэй тохиолддүүд
субботыг багасгах
- file, eof() → багажа гаруу
get파 corn xbl eof.

of stream.

File LL 123 LL". " LL "abc" << std::endl;
представляет собой строку, выводимую корректно.
Приведённого отображения не:
int a; → где а значение самого регистра
"abc" → где а значение д регистра.
→ But вывод → тоже все где а
значение изменяется на бинарный
(на Binary)

В краина със една фамилия представява
последователност от ~~автори~~ ~~автори~~ ~~автори~~ ~~автори~~.

getLine(<имя файла>, &нагугет) -> зерте жо 'ln' жаңы
get() - зерте 1 санды
>> - зерте жо 'ln', 1, eof. ->
б) жадиң және нөхандың түрлерін озакташтырып.

режим на работата: ~~file~~ fstream file ("име"), ~~(режим на)~~
~~работа~~
знач.

Режимът е число от степен на 2 и до 1.: 00001

Вместо да се пишат числа има
надови троичници:

std::

- 1) ios::in - 000001 - Отваря файл за четене
- 2) ios::out - 000010 -> Отваря файл за писане. Ако върху него е свидетелствал, той се затваря. Може да се използват произволни места
- 3) ios::ate - 000100 - Отваря за писане и може рут-указателя на края. Може да се използват места
- 4) ios::app - 001000 - Отваря за писане, след като рут-указателя на края, но не може да се използва.
- 5) ios::trunc - 010000 - Ако файлът е свидетелствал, съдържанието се изтрива.
- 6) ios::binary - 100000 - Отваря файла в двоичен формат.

Числото се ~~може~~ до конфигуриране на режимите с 1. ~~без~~ ~~без~~
такива; ~~зато~~

пр. ofstream file("abc.dat", std::ios::trunc |
std::ios::binary);

Функции:

file.eof() -> bool, дали ами стигнали края на файла

good() -> да ли операциите извършени са успешно

fail() -> дали последната операция е била успешна

bad() -> дали има инаят оп. е бил успешен

clear() -> изчиства последната се е провалила

close() -> затваря потока, когато да хвърли грешка.

Отваряне на файл и изчисление на размера.

```
int getFileSize(std::ifstream& file){  
    int curr_pos = file.tellg();  
    file.seekg(std::ios::end);  
    int size = file.tellg() - curr_pos;  
    file.seekg(curr_pos);  
    return size;  
}
```

seekg(поместване на курсора)
file.seekg(0) -> курсор се изчарва на
поместеното място
(начало),

file.tellg() -> връща число
(текущото положение
на курсора),

```
int getLinesCount(std::ifstream& file){  
    int count = 0;  
    int curr_pos = file.tellg();  
    file.seekg(0);  
    while(!file.eof()){  
        char temp[1024];  
        file.getline(temp, 1024);  
        ++count;  
    }  
    file.seekg(curr_pos);  
    return count
```

III

Двойни файлове. Big Endian

Четене и записване на масиви и обекти

3

При откриването на файла задавате редица на работата binary.

Използвайте функциите: ~~file~~ .read(char*, memBlock, size_t size),
.write(const char* memBlock, size_t size).

Функциите приемат `char*`, което значи, че ако ~~file~~ работи с обекти или данни, като не са от тип `char*`, трябва да се превърнат в `char*`.

Пример: `int a = 150;` $150 \rightarrow \boxed{1110010}$, записва се $\boxed{1110010}$
`file.write((const char*)&a, sizeof(a));`
`int b;`
`file.read((char*)&b, sizeof(b));`
// При четене трябва предварително да инициализира и да е масивът разпределен.

Записване на структура

Без фин. пакет

```
struct A{  
    int a;  
    bool b;  
};
```

~~file.write~~

```
file.write((const char*)&obj, sizeof(obj));  
file.write((const char*)&arr, 3 * sizeof(A));
```

Масивът за запишане често
има по-голяма дължина
от 1, като предвидено
използване на целия

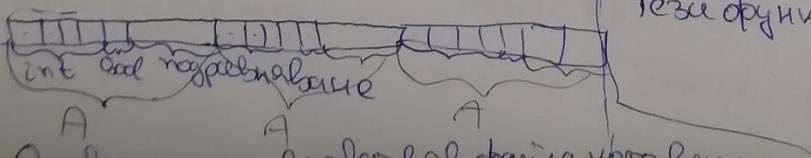
му.

A.read;

```
file.read((char*)&read, sizeof(A));  
A* ptr = new A[sizeof(fileSize) / sizeof(A)];
```

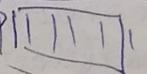
```
file.read((char*)ptr, fileSize);
```

Всичко предварително сме
измерили големината на
файла.



С фин. пакет

```
struct st{  
    char* name;  
    int age;  
};
```



Ако използвате предвиден
метод като записвате пакета,
а често има същия масив.
Задача преда и него записвате
пакетната му и носи имено.

```
file.write((const char*)&st[0].name, sizeof(int));  
file.write((const char*)&st[0].name, strlen(st.name));
```

Зададете имена на всички

```
int size;  
file.read((char*)&size, sizeof(size));  
&char* name = new char[size + 1];
```

```
file.read(name, size);  
delete[] name;
```

За масивът трябва да има
тези функции, ~~които са специални~~

```

struct St{
    char name[10];
    int age;
};

void saveToFile(const St& st, std::ofstream& file){
    file.write(name, sizeof(name));
    file.write((const char*)&st, sizeof(st));
}

void saveArrToFile(const St* pts, const size_t count,
                   std::ofstream& file){
    file.write((const char*)ptr, count * sizeof(St));
}

void readFromFile(std::ifstream& file){
    St obj;
    file.read((char*)&obj, sizeof(St));
}

void readArrFromFile(std::ifstream& file){
    int curPos = file.tellg();
    file.seekg(0, std::ios::end);
    int fileSize = file.tellg();
    file.seekg(curPos);
    St* ptr = new St[fileSize / sizeof(St)];
    file.read((char*)ptr, fileSize);
}

```

// Ошибки исправлены в коде выше

```

struct St{
    char* name;
    int age;
};

void saveToFile(const St& st, std::ofstream& file){
    file.write((const char*)&st.name, sizeof(st.name));
    file.write(st.name, strlen(st.name));
    file.write((const char*)&st.age, sizeof(st.age));
}

void readFromFile(std::ifstream& file){
    St obj;
    int size;
    file.read((char*)&size, sizeof(size));
    obj.name = new char[size + 1]; // Ако не е
    file.read(obj.name, size); // дамашка и
    file.read((char*)&obj.age, sizeof(obj.age));
}

// За макбу якъму иephure dogukum

```

IV

Член-функции. Конструктори и Деструктори. Извикване.

4

Конст и Дестр. при компилация.

Модификатори за достъп. Конструция.

Конструция - отдеяне на кода в класа / структурата на

отделни места. Могат да съдържат private, public и protected

public - всички извън класа имат достъп до първото коя (функции и
private - ~~единствен~~ всичка достъпно е само в приватни)

protected - достъпно е само в рамките на класа и наследниците

наследяване

Така може да се ограничи достъпа на потребителя към кода

Член-функции. → функции, които са част от класа.

извикват се чрез обектите на класовете! obj.function();

Те работят с член-дадените на класа. Могат да са private (извикват се ^{могат да са} само в рамките на класа), public (могат да се извикват отвсякото) и protected (могат да се извикват също в рамките на класа).

class A{

int a;

public: const

void print()

std::cout << a << endl;

Ако функцията наименува член-дадените, тя се маркира като константна, за да може да се извиква от ^{същия и const в хлязги} константни обекти

}

obj;
obj.print();

Член-функции, които са извършват съвместно от конструктори: constructor, copy constructor, operator=, destructor

Constructor: извиква се при създаване на

обект. Автоматично се създава дефолтен, такъв, който не приема аргументи, но може да се дехарактира такъв, който приема. Такъв тип се нарича конструтор, осъществяващ и то не се преопределнява.

- Най-често на бързите, която същото имене като класа.

Не се загава

В него се разглежда обект по някакъв начин, спрямо член-дадените. Конструктора може да приема параметри.

Destructor: извиква се при унищожаване на обекта! В крайна сметка на функцията или просто скончва за нея в stack-а и при извикване на delete за нея в хът блок-а. Най-често на бързите и имена същото имене като класа с "destr" в същия стринг. Destr. Не приема параметри.

struct A{

int a;

int a=0;

A() {a=0;}

A(int a){this->a=a;}

~A(){cout<<"destroy";}

};

при обявяване на A obj; ~const A* ptr=new A(2); ~destr за ^{обект} по референцията на променливата this. Не се вижда copy constr. ини constr.

~destr за обект в стека.

Ако имаме следната функция: void F(Ad obj){}

То можем да я извикаме по следния начин: F(5), тъй като има такъв конструктор (които приема число). Това е конвертиране. Можем и да го зададем, като напишем "explicit" пред конструктора.

Композиция: Ако имаме генерации други обекти, то конструкторът

трябва експlicitно да извика нужния конструктор на мен-даниите. По подразбиране се извикват default-ни.

Struct A;

Struct B;

Struct X{

 int a;
 A objA;
 B objB;

X(): a(), objA(), objB()

Тук се извикват defaultни

значения се извикват в
наред от конструктора на X.

Констр. на X се извиква след това. constr of X.

X(int a, int b, int c): a(a), objA(b), objB(c)

Тук се извиква констр. с параметри int.

X()

 ~objX

 // при инициализация на фун. метод.

 1) ~dest of objB

 2) ~dest of objA

 3) ~dest of a.

В constr: Първо се построват мен-даниите
после X.

В dest: Първо се унищожава X, после
мен-даниите в обратния ред на
създаване.

X arr[5];

X* ptr = new X[5];

Създава се масив от 5 обекти на X и
се извикват default-ният и едни стр.
За масива default const. е загубен.
Иначе и този масив не може да се
създава.

get-функции -> Задължителни. Редът даден до инициализацията
мен-даниите. (някои поне)

set-функции -> променят мен-даниите (чрез некои условия или
може да предадат валида-
чески)

Буфера X:

int getA() const { return a; }
void setA(int newA) { a = newA; }

V

Разделна компиляция.
Копиране, конструектор и оператор=

5

Кодът в една C++ програма може да бъде разделян на много файлове, които се компилират независимо един от друг.

Компилират се само .cpp файловете, след това за всеки се свързва обектен (.obj) файл и след това обектните файлове се свързват в .exe файл, който е финалния.

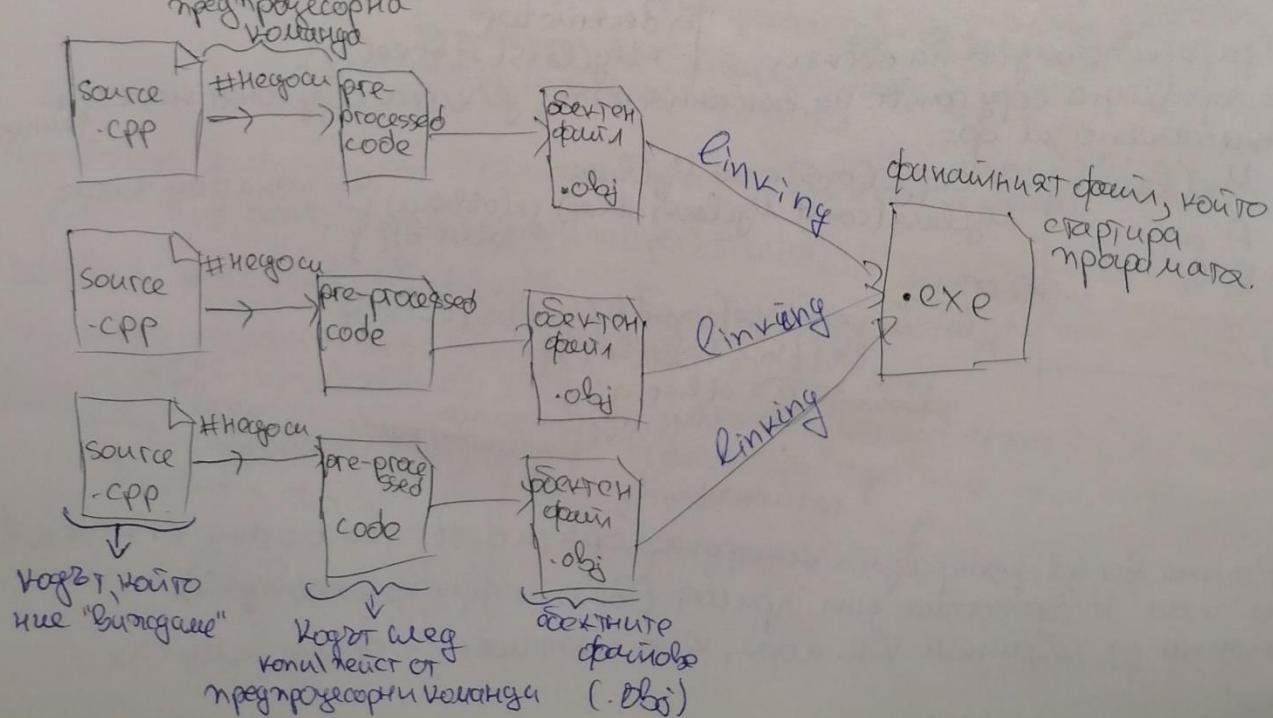
За да се използвате начинанието от разделна компиляция, трябва всички членове (може и клас и функции) в .h и .cpp файл. В .h са ~~такие~~ съдържанието на класа и методите, а в .cpp е имплементацията на методите.

Ако искаш да използваш даден клас и якоже в програмата, е нужно само да инклудиш .h файла:

#include < "имен. h файл" > - библиотека . # е предпроцесорна команда, която

се изпълнява преди компиляция. Това която #include прави е да сарфасте кода от файла, който инклудаш, на реда, когато не си имаш #include. Триумфълът на този се постига кога от файла, който формира финалния код, който все още се компилира.

На своята практика работи и #define what with, който възпроизвежда че замени what с with.



→ приелце се за конструктор, ако и да не е ~~defaut~~ defaut.

Основен конструктор и деструктор, този създават автоматично и
Копиращ конструктор: MyClass(const MyClass& other) - създава нов обект
оператор =: MyClass& operator=(const MyClass& other) - модифицира вече
съществуващ обект да е идентичен на other
↓
функция ↳ Въртида редовречници, за да
може да генерирате.

class MyClass {

const MyClass(const MyClass& other){}

MyClass& operator=(const MyClass& other){}

};

MyClass a; // default constr.

MyClass b(a); // copy constr.

MyClass c = b; // copy constr.

b & c = a; // operator=

MyClass* ptr = new MyClass(c); // const.

void f(MyClass a) → При извикване на функцията, тя се вижда как copy constr.

void g(MyClass a) → Няма да се извика как copy constr, гори и дестр.

Дефинираните от компилатора копиращи constr/op= работят предвидно,
ако в класовете има функцията move. Ако и да, трябва също
да предизвикате copy constr/op= / dest. За тези създавани
private глен-функции free() → трябва да имат copyFrom(const MyClass& other)
Логично да имате.

Тези функции са използвани, когато трябва бързо да се извикат.

MyClass {

int a; // за примера това
егълът, а не масив.

private:

void free()

delete a;

void copyFrom(const MyClass& other){

 a = new int(other.a);

};

⇒ Копиращия конструктор:

MyClass(const MyClass& other){

 copyFrom(other);

⇒ оператор =

MyClass& operator=(const MyClass& other){

 if(this != other){ // ако все пак има
 free(); // своя обект
 }

 copyFrom(other);

 return *this;

⇒ деструктор

~MyClass(){ free(); };

При използване на обекти

е правилно copy constr. на външния клас да извика copy constr на едн. пакет.

Analogично за op=.

MyClass {

A a;

B b;

y;

⇒ copy Constr на MyClass:

MyClass(const MyClass& other): a(other.a), b(other.b){ // единично пакет.

⇒ oper.=

MyClass& operator=(const MyClass& other){

 if(this != other){

 a = other.a;

 b = other.b;

 } // единично пакет.

 return *this;

Различава се копиращия конструктор, op= и dest., също ако бързата
чи и да необходима едн. пакет (бърз. и при използване).

Монтирайте го за бързото K.K. и op=, като напишем "= delete;" или так.

VI

Болячите разборки

6

1) default constr. → създава се обрт, ако не има ~~изписани~~ наименования.

2) copy constr → създава се обрт, ако не има изписано наименование

3) operator= → създава се обрт, ако не има изписано име

4) destr. → -||- не има изписано име.

Тези, които се създават обрт от компилатора, работят коректно, тъкмо
във всичките им съществуващи случаи. Таска избягва и да съмнение
и никој не ги разглежда.

Ако има син. на името, то поведението на копирането е грешно.

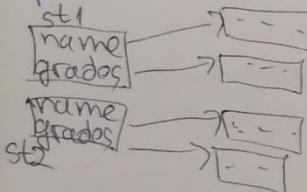
`class Person {
 Student name;
 int grades;
 int size;
}`

Definiraniите от он компилатор K.K. и он = съдържани
наименования:

`name = other.name;`
`grades = other.grades;` > T-e!

Също дестр. не дава оговорка за името

При работата със структурите съдържани:



За тази причина
необходимо е правилно изписани функции free() и

void free() { delete[] name; grades; }

void copyFrom(const Student& other) {

size = other.size;

grades = new int[size];

for (size_t i = 0; i < size; ++i) {

grades[i] = other.grades[i];

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

free();

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

VII

Приятелски класове и функции. Преопределение на оператори.

7

Приятелска функция на един клас е **външна функция**
за клас, който има достъп до ~~т~~ генериранни на класа.
~~Общ~~ Една функция се обозначава като приятелска в
същия клас. Аналогично важи и за приятелски класове.

class Example{ // Декларират се всички членове на класа
 friend void examplefunc(); // friend няма крака / функцията
 friend class Example2;
};

Тїedefониране на оператори.

Операторите $\$ab\alpha$ чието (1 аргумент) или двойници (2 аргумента) има и тернарни.
 Характеризират се с асоциативност \rightarrow лявосочищавни ($(a\$b)\c)
 \hookrightarrow дясносочищавни ($a\$(b\$c)$)

Характеризират се и с приоритет и с посочен ~~специален~~ специален приоритет

- ↳ постфикс сен а\$ → имеє думки
- ↳ инфикс сен а\$ в операції

В C++ операторы представляются функциями

Може да предадирише също поведението на физен оператор.
Т.е. не може да предадирише асоциативността, приоритета
или изучаваща спрайт-организация

Class Mums

int a friend std::ostream& operator<<(std::ostream& stream, const Num& arg); } class Num { public: Num& operator+=(const Num& other); } // Klasa.

a += other.a;
return *this;

```
std::ostream& operator<<(std::ostream& stream,  
                           const Num& arg){  
    stream << arg.a << endl('n');  
    return stream;
```

友谊
↓
Това е ~~Ф~~ извън класа. Може да е в ^{човека} ~~човека~~.
Трилементната на функцията
~~се~~ се пише friend.

В тези 2 случаи възникват
ответните референции,
за да може да генерират
неколко поддни възможности
на оператора (единично или)
на 1 ред.

VIII

8

Масиви от указатели към обекти.

Използване. Работа с използване.

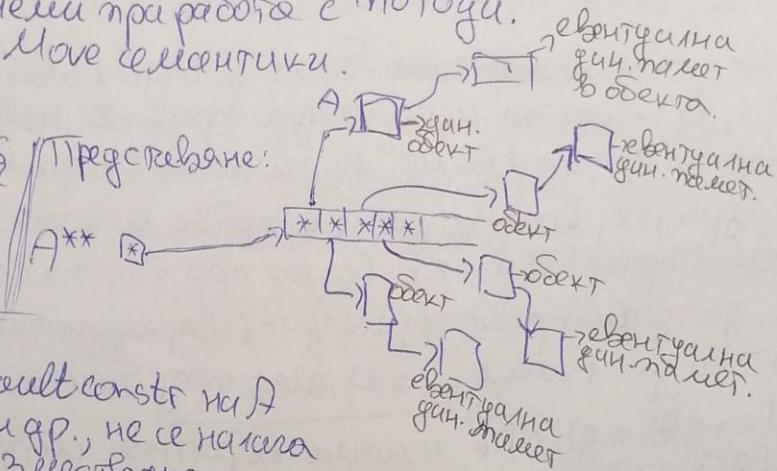
Проблем при работа с потоци.

Моя семантика.

class SomeCollection { Представление:

A** arr;

и



Позвал не е нужен default constr на A

- при сортиране и пр., не се налага

копиране, се разделят върху обекти

- възможна е празна къмп с nullptr.

осигури:

- delete[] не е достатъчно - то трябва да има указатели към обекти.

преди това трябва с усъвът да се изтрият и обектите

Използване! Или да сигнализирате, че нещо не е наред с програмата. Трябва се чрез: throw new cur. Може ли да throw нова идентична стойност или обект (от exception). Ако не се обработи този грешка, програмата трябва да обработи този блоц. Обработва се с try/catch блок.

ногийкото може
да хвърли грешка.

Catch (Липу)

нечуйте си за
модели и if else
справо със грешка
е грешката

* Когато създавате пред
създавате.

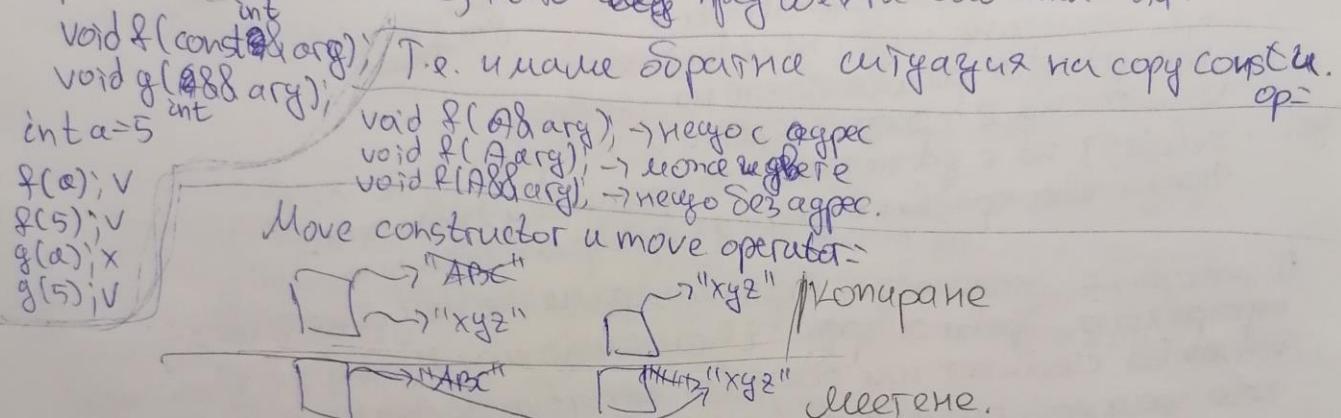
то при throw, функцията трябва да извика функция, ...,
важат се деструкторите от този функция и
грешката предизвикава да се даде наследен
по върху този функция, докато не нападне
try/catch блок. Този метод извиква деструктор на
обектите. Ако такъж се хвърли грешка по този
или иначай деструктор хвърли грешка, то тя не може да бъде
обработена. Може ли да обработвате със 1 грешка. Задава
се в деструктора. Не хвърляйте грешки. Задавайте потоци в дефиниции
като се използват .close(), което функцията може да хвърли грешки.
Задава е добре със създаване потоци и .close() да
е в един try/catch блок, а останалата предишният ред от
кога създава, за да се обработят грешките.

Move семантика

lvalue \rightarrow bc., за което има място б конст (агрес)
 rvalue \rightarrow bc, което не е lvalue (математически член).

В ~~Macobe~~ move семантика се използвали в определени моменти, в които не искаше да прави нови копия на данните. Например като подаваше математически член в класовете, като имат агрес, то вместо да им прави копия, директно си пренасочи пойнтерите им тях. (само при удач. начин).

Може да дефинирате функции (съвръзни), които могат да приемат само rvalue, като ~~arg~~ пред бранга следващата 88.



MyClass(~~const~~ MyClass&& other){

name = other.name; Move constr за \Rightarrow
 other.name = nullptr;

MyClass& operator=(MyClass&& other){

if(this != &other){
 free();
 name = other.name;
 other.name = nullptr;

return *this;

MyClass{
 char* name;

Нека имаме MyClass A (без дефинирани move семантици) и MyClass B с дефинирани move семантици.

A f(){

A a;

return a;

B g(){

B b;

return b;

}

A obj1;

B obj2 \Rightarrow по конст

obj1 = f();

move operator =

obj2 = g();

move constr.

(създава две конст

Бързоначно създават конст, които са създади

Класът Копирайторът сама реалства да си извика move функциите. Ако искаш да го създеш, използвай `std::move()`. Така заставяш да използва няма да създъши конст и да извикаш бекра и ще се извикат move функции в класа.

IX

Конвертиращ конструктор Композиция и сървегауза. Шаблони.

9

Конвертиращи конструктори са тези, които преенят един аргумент. Така, когато една функция приеме обект по копие или по константна референция, може да има и изпълнителен аргумент. Така се създава временен обект и в края се унищожава.

Class A{

int a;
public:

A(int a){this->a=a;}

}

```
void f(const A& arg);
void g(A& arg);
void h(A arg);
```

f(5); v

g(5); x → създава нов обект.

h(5); v

При композиция в клас, в който има динамична памет, то конструктора на членния предиква да извика конструктор на член-данични обекти (това предиква всички да стane несъмнено член динамични/памет).

Всички констр. трябва да извикат конст. на член-даничните обекти.
Опер.= предиква всички опер.= на член-даничната обект.

Рестр. предиква да всички реагират на член-даничната обект (стъкло обкт.)

Class A{

int *arr;

~obj();

Приема и, че
се дефинирани
free и copyFrom.

~A(){free();}

⇒ конст констр

A(const A& other): obj(other.obj){

copyFrom(other);

3 A& operator=(const A& other){

if(this != &other){

free(); obj = other.obj;

copyFrom(other);

4 return *this;

не изпълнено да са
викане е компликуван

free и copyFrom
се групират само
за дин. памет и елементарни
присъствици член данни.

Шаблони са функции/методи, които работят с обобщен тип данни.
Много десе е вс. Унифицирана се template < типене > (използва се преди класови и преги формул.)

При композиция констрактора конст/предиква ако има шаблон когато замества
шаблон с конкретна, която ползваше. Констрактор, който замества се в
шаблона, предиква да са дефинирани. Ако пази и масив, предиква и
default констр., иначе T**. Пишат се в едни обекти. Кръг един, защото иначе copy копира констрактора, започнат си. К и. и кръгва,
зато предиква да е конст/нейстче, а тогава копира констрактор.

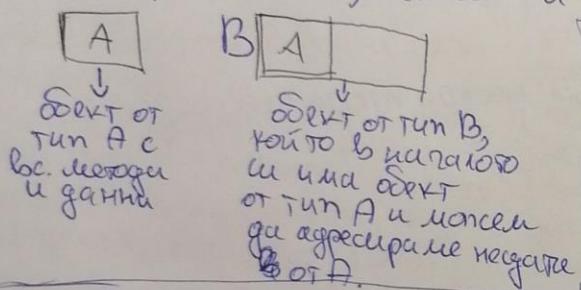
X

Наследяване, Възможни параметри на функции
Казатели, референции
Конгр. Деср., Копиране.

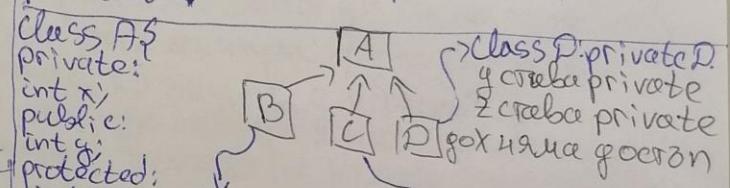
10

Наследник клас е клас, който спада към клас с икономически
съдържани съдържани клас, който в него може да се добавя и
допълнителна. Наследяване се използва, за да не се пише
пъвично умия обикновено за да се сворят логически
класове един

Класът B ще получи достъп до членове от външните на A.
Така членът от A е и в B, и във, и това членът може да се
прави с A, може и с B. Типът на наследяване:



class B: типа на наследяване > A/иначе
така членът от A е и в B, и във, и това членът може да се
прави с A, може и с B. Типът на наследяване:
private: → default при класът
public: → default при struct
protected: →



Наследването, което трябва и градиве
да има логика, не може

Да се прави наследник на Person

class A;
class B: public A;

A a; ✓
B b; ✓
A&a = &a; ✓
B&b = &b; X
A*ptr = &a; ✓
A*ptr = &b; ✓
B*ptr = &a; X
B*ptr = &b; ✓

void f(A&arg); void g(A*&arg); void h(B&arg)
f(a); ✓ h(b); ✓ f(b); X g(a); ✓ g(b); ✓

Пътищата референции
от базовия клас към наследника
работят, защото базовия
и само тези са в наследни.
Обратното не работи, защото
наследника не гърди се

си изчеза.

Тъй като в началото на наследник има базовия
клас, то трябва да го преграви по подобно на конструирането
конструктор на наследник трябва да подгответ родителя си.
Копи констр. на наследник трябва да копира и родителя си.
Оператор на наследник трябва да копира и родителя си.
Копи констр. и оп. недика работили правилно (тези дефинират
от конструирането), ако в наследника има констр.
(референции).

Конструирането дефинира!

От наследник клас можем да направим първия наследник
(референция), но че можем да извикваме само
методите на наследника клас. Базовия клас възпроизвежда само съдържанието

/// У деструктора трябва да се извика конст. на базовия, заради редовреждащите към наследник
B (const B& other): A(other){
 к.к. и A не адресира само членове
 A в Bother.}

3. /// в кл. на B

```
B::operator=(const B& other){  
    if (this != &other){  
        A::operator=(other);
```

 // free, copyFrom...
 return *this;

Ако имаме следната ситуация:

```
B obj = new B();  
A* ptr = &obj;
```

можем да адресираме само членовете на родителя и единств. virt. функции.

т.е. ако направим delete ptr;, то

ще се извика дестр. на A, и можем да имаме член от B, която да не се изтрива. Задава

но това е към следващите теми.

Деструктор на наследник извиква деструктора на родител.

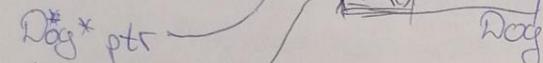


XI

Полиморфизъм.
Статично и динамично
съвръзване. Абстрактни класове

11

Статично съвръзване
– по време на компиляция
class Animal {
 void roar();
};
class Dog : public Animal {
 void roar();
};

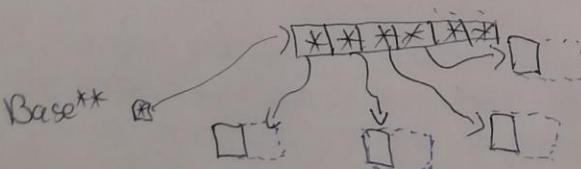


`Dog* ptr`
`Animal* ptr2`
`ptr->roar()` се извика този на Dog.
`ptr2->roar()` се извика този на Animal.

Всички поинтър са извикани, този който "випада" `Animal*` не предполага, че съдържа всички `Dog::roar()`. Т.е. извиката функция се определя от типа указател по време на компиляция

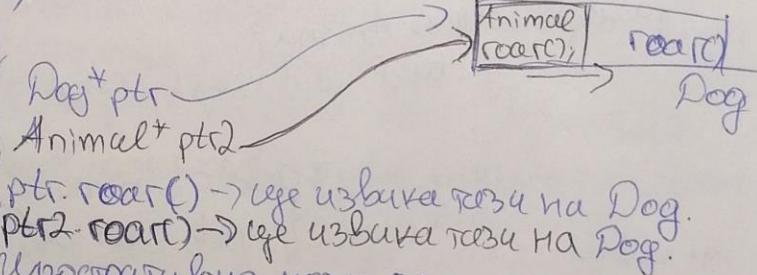
Виртуалните функции ни позволяват да свържем място от указатели към базовия клас, който могат да съдържат към различни наследници. Същите можат да съдържат пребърнатите виртуални функции и съдържат наследници

`Base**` са такие правилни `destr.` на наследника, който от своя страна съдържа този на базовия клас.



Динамично съвръзване
по време на изпълнение на програмата

```
class Animal {  
    virtual void roar();  
};  
class Dog : public Animal {  
    void roar() override;  
};
```



`ptr->roar()` → се извика този на Dog.
`ptr2->roar()` → се извика този на Dog.

Илюстративно може да си по представяме така: `ptr2` приема функцията в класа `Animal`, но защото е `virtual` се погне да работи ~~се~~ наследниките, ~~което~~ ~~което~~ не намира последната `override`-ната функция `roar()`

Това не се случва така, също се грее виртуални гадливи. В наследник си има гадливи с виртуални функции, които са преопределени, и спомената поинтър определя по ней коя функция да извика.

Т.е. дин. съвръзване се изпълнява с помощта на вирт. гадливи по време на изпълнение на програмата на изпълнение на програмата

Така не се прересуващ от вид на наследника, заради виртуалните функции.

Виртуалността на функцията е наследявана от тя е `virtual` в `base class`, то тя е `virtual` и в наслед. гори да не се тиши.

Абстрактни класове

pure virtual function - предназначена е за се override-не. Всички наследник. Обозначава се в базовия клас като челика virtual функция се инициира = 0. И нямат ги и не може да се вика от базовия клас, засега тој става абстрактен. Ако има поне една pure virtual function в клас то той става абстрактен, не може да се правят обекти от него, само пойнтери и ресурси. Предназначен е за наследяване. У наследник трябва да override-не тези функции, иначе остава абстрактен.

Пример: class Animal {
 virtual void roar() const = 0;
};

class Dog : public Animal {
 void roar() const override;
};

С оглед на обозначава
рефлексията е предопределен
от Академията. Има, не се
смърт и обозначава, че
това последното се предефици-
раче в иерархията. Идеята
да се използва, че
наследства.

Полиморфизъм: едно име на функция, много поведения.
Реализират се чрез virtual функции, които се предефинират
по различен начин и чрез указатели от базовия клас се
извикват правилните, без да се интересуваш коя какъв наследник

Прилага се в класове с общи прародители, чрез виртуален метод.
Предефинира се или не този метод. Активира се с указатели от
базовия клас. Чрез свидетелски указатели и приема #обекти.
Затова деструктор е virtual, иначе указателя се изтриват
"седеци" в наследника.

XII

Варианти Години.
Колекция на обекти в
полиморфна иерархия.
Factory pattern.

12

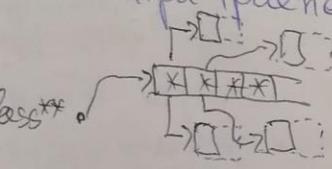
Полиморфен контейнер представява масив от указатели от тип базовия клас (~~не са абстрактен~~), който съдържа към един обект на наследниците, като не се интересуващ за какви са. Челта е да можем да извикваме биргудалните функции, ~~не разделяни~~ грез функциите в класа на колекция. Още

Отново искаме, при добавяне на обекти в колекцията да се изздава нов обект за всяка колекция. За тази базовия клас изздаваме риге virtual clone функция:

virtual BaseClass* clone() const = 0; // връща динамичен обект
clone на събери

else

При трасе грешка:



Број със чакви да викнем delete на всеки обект, грез virtual destructor опасност от деструктор на памет

Задача за извикване delete за динамични указатели.

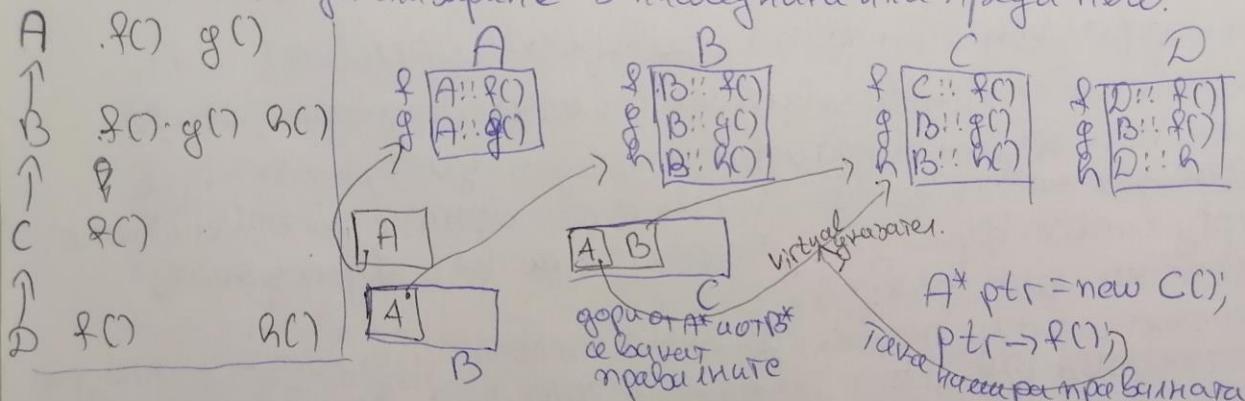
Контейнерът си навсякога има вектор, и се resize-ва като вектор.

Factory Pattern: design pattern за построяване на обекти по начин на изграждане:

Представява клас ~~създаден~~ се от функции, които ~~се~~ взимат информация за обектите от иницире или че се подава. Или се правят и валидации. Връща се обект по копие или ~~не~~ се изздава. Тези чакви функции са обекти, засърди от служба. Има съанс от грешка на място при връщане на указател ~~или~~ дин. обект, грешка за се извиква при добавяне на обект в колекцията: ~~внимава~~.

1. Factory class-a да се използва извън клас, да се построи обект, да се подаде като параметър на addfunction и да се извика clone.
2. Factory class-a да има в колекцията, ~~и~~ упомянете данни за построяване. Не може обект да се подават на add функцията, грез ~~този~~ Factory то да се построи и да се добавят. (тук има и без clone).

Виртуални таблици: За $\&$ наследник на клас се създават таблици, в които се поставят ~~всички~~ предефинирани
коя е последната ~~предефиниция~~ на $\& virtual$ функция,
която е инициализирана в наследника или преди него.



Ред.: Указателя (ptr) наимука правилната вирт. таблица (в случая за C),
и от тази правилната функция
~~ptr->f(); вирт. функция -> правилна -> вирт. таблица.~~
~~ptr->f(); вирт. функция -> правилна -> вирт. таблица.~~

Нама запазена структура и начин на работа на вирт. таблиците. И коями-
то може да ѝ реализира по разл. начин. Горното е ~~один~~ общи структура
и начин на работа. Вирт. таблица не са общи, ~~зато~~ ^{зато} засилват се.

XIII

Type Casting. Многостепенно наследование

13

1. const-cast: $\text{const } A^* \text{ ptr} = \text{new } A();$

$A^* \text{ ptr2} = \text{const_cast}\langle A^* \rangle(\text{ptr})$
- макс константността
на пойнтар

първи израз.

- често само ако const pointer-a, който максимуме константността,
кори към обект, който някога не е бил константен,

2. dynamic-cast $\text{dynamic_cast}\langle \text{първи} \rangle(\text{израз});$

- преобразува към първия тип и прави проверка за валидност

- ако преобразуването не е валидно, то новият пойнтар ~~се наследява~~ ще бъде nullptr.

$A^* \text{ ptr} = \text{new } X();$

$X^* \text{ ptr2} = \text{dynamic_cast}\langle X^* \rangle(\text{ptr});$ // константно

$Y^* \text{ ptr3} = \text{dynamic_cast}\langle Y^* \rangle(\text{ptr});$ // nullptr. ptr не сори към
обект от тип Y

$A^* \text{ ptr2} = \text{dynamic_cast}\langle A^* \rangle(\text{ptr});$ // не може да бъде nullptr

3. static-cast. $\text{static_cast}\langle \text{първи} \rangle(\text{израз});$ // загоди не е правилен, затова хвърля грешка (try/catch)

-> същото като dynamic-cast, но не прави проверка за валидност.

- затова е по-добро

- прави ли също също сигурни, те е валиден cast-банер

4. reinterpreted-cast

- пойнтар от която е тип към пойнтар от която е тип

struct A {

$A \text{ obj};$

$\text{int } a;$

~~int * reinter~~

$\text{int } * \text{ ptr} = \text{reinterpreted_cast}\langle \text{int }^* \rangle(\text{obj});$

$\text{char } b;$

~~char * reinter~~

$\text{char } * \text{ ptr2} = \text{reinterpreted_cast}\langle \text{char }^* \rangle(\text{ptr});$

$\text{bool } c;$

~~bool * reinter~~

$\text{bool } * \text{ ptr3} = \text{reinterpreted_cast}\langle \text{bool }^* \rangle(\text{ptr2});$

Записани са
един го фрага
б наследи.

и нещо
пойнтар.

5. C style cast : $B^* \text{ ptr};$

$A^* \text{ ptr2} = (\text{A}^*)\text{ptr};$

> 1. изброя прави static-cast -> все още ако се опита се да наследи
база ще излезат към наследници.

> 2. прави dynamic-cast.

> 3. направя прави reinterpreted-cast.

Множествено наследяване

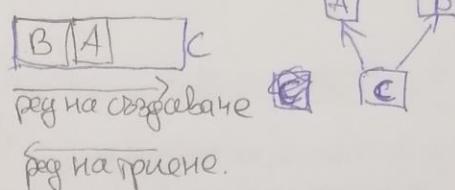
Кога 1 клас наследува няколко от 1 клас:

Ресурсната на C: ~C{

 ||C stuff
 }~A, ~B ||C stuff.

Конкр. на C: C(): A(), B(){}
 {~B на B } заради
 {~A на A } предварителна наследяване
 дестр. съвт. на обратно

class C: public B, public A {};



reg на свързане

reg на промене.

copyFrom(other);

Хоти Конкр.: C(const C& other): A(other), B(other){ C stuff }

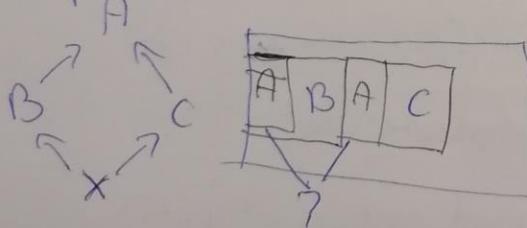
Copy = : C& operator=(const C& other){
 if(this != &other){
 A::operator=(other);
 B::operator=(other);
 free();
 copyFrom(other);
 }
 return *this;

Представлява сърцето
не имате обект от
типа A и от типа B
запъти данни.

Горните 2 се правят от компилатора. Ние ги разделяме, ако
в наследника (C) има доп. наслед.

Динамичен проблем

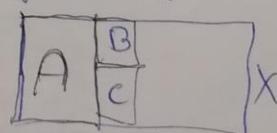
проблем



Ако родители чисто базират на природата, то в X ще има 2 инициализации на A.

решение

Виртуалното наследяване
се прави от наследник
да има една и инициализация!



Това B и C ще спадат един
общ обект A.

Заради виртуалното наследяване
B и C не викат констр. на A, затова
трябва да се извика в X

X вика констр. на B
Конструкуйте с елементи на
виртуални класове, трябва да
извикват констр. на всички родители
не само тяхните.

X(): B(), C(), A(){}
 =====

Създаване на абстрактни клас, от който и се построят дървото,
в който имае функции за копиране, virtual destr и евентуално
други virtual функции, отговорящи на мякото полиморфно
действие спрямо заданието. (например за мякото представяне).

Родовия клас ще наследява все и образуващите класове
за мякота на дървото (те назват информација за себе си),
и колкото класове са нужни за състоянието върхове (в тях
тези са базова паметри, съседи или подмножество, или кояли ~~помощни~~
мякоти други върху).

Destr. на мякот си трябва да е динамични
иначе дан. памет не е
必需но разчистване.

Destr. на всичко друго, викате delete на паметите и
прие вс. данни. памет, ако е нужно.

А. К и Апор- спрямо необходимостта.

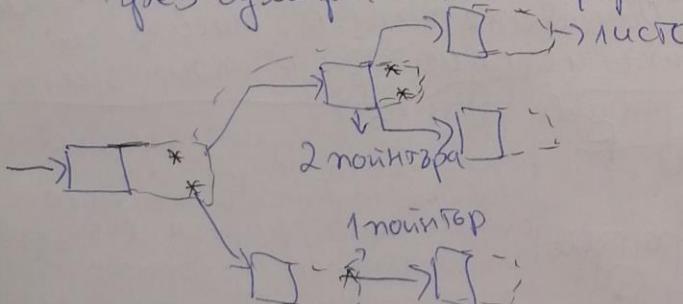
Функцията за ~~изменение~~^{да} преминаване в листо, в реда същата
спрямо данните си.

Функцията за преминаване на всяка квадратна област същата
от паметите.

Пазване на мякота информация или бройка в място;

Пазване на мякота информация в същаклас.

Пазване на мякота информация или бройка на всяка квадратна област
чрез съхранение на мякота информация от паметите.



Викане функции в първия
памет или мякота член
мякота информация за мякото, то
рекурсивно викане на мякотите
функции, докато не стигне
листо.

При мер: логически изразъ, от була $((P \vee Q) \wedge T)$
представяне

Класове: Boolean Expression - базова кл. Тази и риге virtual
функции clone и evaluate, си виртуален десктруктор и бројка
на буквите символите, като участват в израза, в булов масив
от 26 символа и всяко буква участва на своя индекс
изменение true (при създаване).

Evaluate връща буловата стойност и приема масив от 26 символа,
така че стойностите на символите (базирано на всички групи).

Клас за чистото: Тази същ за символ, който предоставява.

Destr. не е определен. Конструктора приема символа, в масива за
броят на символи (~~protected~~) прави съдържанието на индекса на true.

Evaluate приема структурата със стойностите и връща стойността
от индекса на символа, които назим.

Clone връща пойнтер към new Variable(~~public~~); //конст.

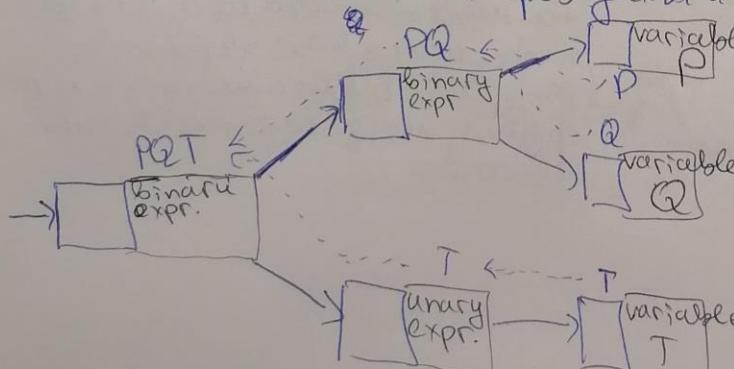
Клас за binary expression: Тази същ за операциите и
2 пойнтира за разклонение на явно и наследено.

Редгр. вика delete на 2ти пойнтира (базова)

Констр. приема два 2 базови пойнтира и спомага този коя да
програмата ни може да има вакнен clone.

Evaluate връща стойността на приемената спомага същ
на операциите и стойността, която връща evaluate на
2ти пойнтира. Clone създава копие на текущия обект (зависи
от конкремтната стойност на пойнтира).

Клас за unary expression: Аналогична binary expression, но че
1 пойнтир. И флага на класа при създаване попълват масива си
за броя на символи чрез учили и спомага ~~реализации~~ от пойнтира.



Изразът е същиг със скоби
около всяка операция.
Използват се твърдки и наследената
стока. Като среднини '()' и брои
като среднини ')'.
Ако броят е 0, то попадащият
на оператор спомага на него
рекурсивно възпроизвежда
отново за вс. до съединения
индекс и така за вс. след него.
Ако константата на съринга
е 0, то това е Variable.

Проверка за Гавиология: Поглеждане на
функцията за парсване им и масивът, който нази чи то букви участват
в интерпретацията. В интерпретацията възпроизвежда чи то
име на стойността на индекса и делум индекса. Това го прави
2 variant чи то и възпроизвежда чи то създадена интерпретация.