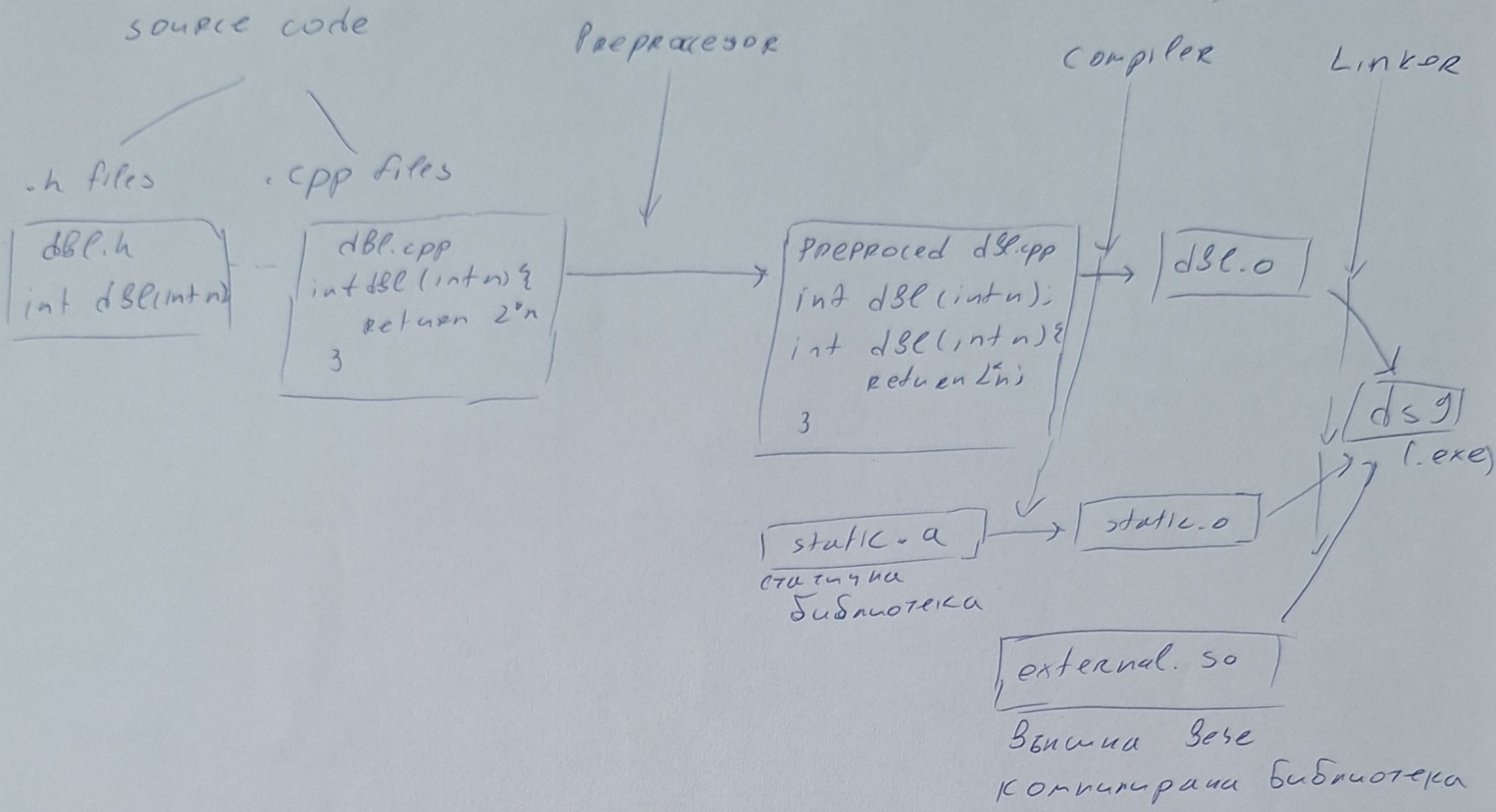
5. Разделна компилация. Процесор. Конст. конструктор и ОП =. Композиция и агрегация



1. Процесор - обработка на стринговe
- код, който се замества с друг код
- специален символ "#"

а) #include "A.txt"
- този ред 1:1 се замества със съдържанието на A.txt

Пр: init.txt
int a=4

main.cpp
int main(){
#include "init.txt" ↓
a++;
cout<<a;
}

б) #define
- макроси
- заместват парче код с друго парче код
- по-бързи от ф-ции, защото са напълно inline

Пр: #define max(a,b) ((a)<(b)?(b):(a))

int c = max(2,p);

в) header guards

- позволява блок от код да бъде изпълнен само 1 път и да му се даде специално име

Пр: #ifndef JOB_H
    #define JOB_H

    #endif

- това може и да стане с #pragma once

2. Компилация

- синтактичен анализ - проверки за грешки a+++;
- семантичен анализ - проверка за ^присвоена^ вярност A obj;
  obj = 73;

- междинна оптимизация
  - O1, O2, O3 - казване на нивото на оптимизация
  if (7>3) - бива махнато, защото е винаги вярно
  (inline на ф-ции понякога)

- assembly code

- machine code (0...1) - всички .cpp се компилират до .o (binary) файлове, компилират се статичните библиотеки (.a)

- разделна комп. - комп. отделно един от друг и така се избягва каждата от прекомпилация на един и същи файлове

3. Linking

- етап на компилация, в който външните зависимости в .o файлове биват решени

- forward declaration - способ, който се използва за да "обещаят", че обекта ще съществува и ще бъде намерен от Linker-a. (решава проблема със circular dependency)

Пр: a.cpp
    #include "b.cpp"
    struct A {
        :
    };

b.cpp
    struct A; ← това е forward declaration
    f(A* obj);
      ↑ можем да използваме само ref и pointer, без конкретни ф-ции и инициализация

- На този етап от компилацията се добавят и всички .lib/.so динамични библиотеки

~~[illegible]~~

- Оптимизации за CPU

4. Output
- получаване на 1 финален файл, той може да е:
  - .exe - екзекютабъл файл
  - .so/.lib - дин. библиотеки
  - .lib/.a - статични биб.

Композиция
- отношение м/у обектите, в което външния клас A "притежава" B и отговаря за жизнения му цикъл.
- B няма предназначение в системата извън A
- A менажира ресурсите (трие, копира, създава) B
-

Пр:
```
class X {
    A a;
    B b;
    C c;
}
```

Агрегация
- отношение м/у обектите, в което външния клас A, не отговаря за вътрешните ресурси на B.
- A не се грижи за (триене, копиране, създаване) на B
- A "използва" B

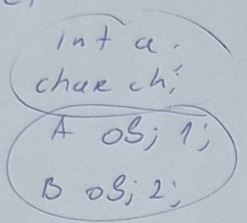
Пр:
```
class Singleton {
    DBprovider* prov;
}
```

Пр: Singleton не трябва да трие DBprovider!!!

Копиране на обекти

Копиращ конструктор
- приема обект от същия клас и текущия клас създава негово копие
- ако няма се генерира автоматично от компилатора при него се викат рекурсивно копи-конструкторите на всички обекти надолу по веригата
- при наличие на к.к. компилатора няма да се създаде def()

Пр: struct X



→ метсру
→ к.к. на А и к.к на В

- това е автоматично ген. от комп.

Пр: A obj;
A obj2(obj 1);
f(obj); → pass by value к.к.

void f (A obj)

Пр: struct X {
int ai;
A a;
B b;
}

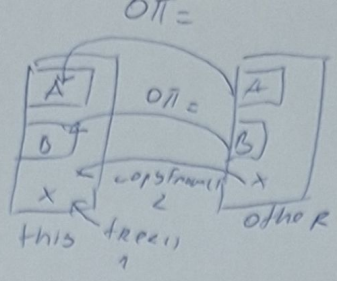
X(const X& other) : i(other.i), a(other.a), b(other.b) {...}

Оператор = / Оператор за присвояване
- превръща съществуващ обект в копие на друг обект
- изчиства ресурсите и следователно прави копие
- автоматично генерираният вид оп= на композираните от обекти
- връща референция към настоящия обект
- дясно асоциативен оператор

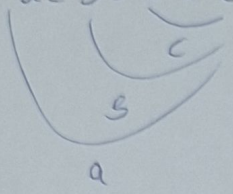
Пр: извикване
A obj 1;
A obj 2;
obj 2 = obj 1;

X

```
int i;
A a;
B B;
```



Пр: a=b=c=d; (дясно асоциативност



Пр:
```
struct X {
    int i;
    A a;
    B b;

X& operator=(const X& oth) {
        if(this != &oth) {
            A::operator=(oth.a);
            B::operator=(oth.b);
            free();
            copyfrom(oth)
        }
    return *this
    }
```

RVO Пр:
```
A oB; //
A oB2 = oB;
```

Тук се извиква К.К. а не ОП=, за да се оптимизира. Тук = е друг синтаксис предоставен от IDE-то:

⚠
```
A create A() {
    return A()
}
A oB = create A()
```

Тук се случва RVO от компилатора и се спестява К.К. ~~това~~ този "трик" използван от компилатора е добре известен optimization