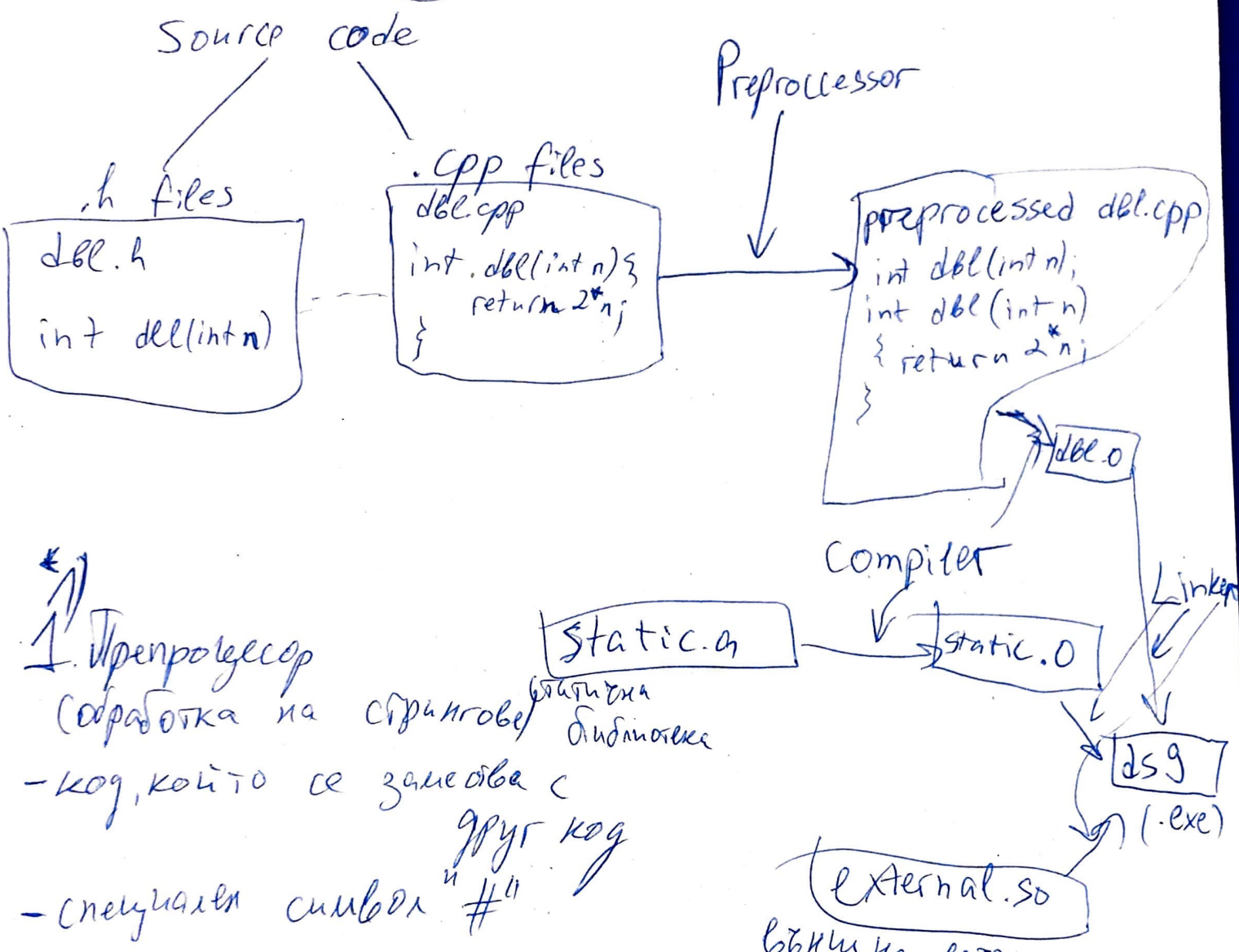
Тема 5 | Разделна компилация. Препроцесор. Конд. констр. и ООП. Композиция и агрегация



1. Препроцесор
Обработка на стринговете
- код, който се замества с друг код
- специален символ "#"

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

Пример:

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

Пример: #define max(a,b) ((a)<(b) ? (b) : (a))
int c = max(z,p);

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

Пример: #ifndef JOB_H
#define JOB_H
;
#endif
- това може и да стане с #pragma once

2. Компилация
2) - синтактичен анализ - проверка за грешки a+++;
3) - семантичен анализ - проверка за смислен вярност A obj;
obj = 73;
4) - междинна оптимизация
- if(7 > 3) - бива махнато, защото е винаги вярно
(inline на ф-ции понякога)
5) - assembly code
6) - machine code (0_1) .cpp се компилират до .o (binary) файлове, компилират се статичните библиотеки (.a)

- разделна компилация - компилират се отделно един от друг и така се избягва нуждата от прекомпилация на цели и други файлове.

*7) 3. Linking

- етап на компилация, в който външните зависимости в .o файловете биват решени и се свързват до .exe файл.

- forward declaration - метод, който се използва за да "обещаем", че обекта ще съществува и ще бъде намерен от Linker-а. (Решава проблема със circular dependancy)

Пример:

A.cpp

```
#include B.cpp

struct A
{
:
}
```

B.cpp

```
struct A; forward declaration

f(A* obj);
```

Можем да използваме само ref и pointer, без конкретни ф-ции и инициализации

- на този етап от компилацията се добавят и всички .dll/.so динамични библиотеки

*8) Оптимизации на CPU (вече .exe е създаден от Linker-а)

1. Output
- получаване на 1 финален файл, той може да е:
  - .exe - ...
  - .so - дин. библиотека
  - .lib/.a - статична библиотека

Композиция
- отношение между обекти, в което външния клас А "притежава" В и отговаря за жизнения му цикъл.
- В няма предназначение в системата извън А
- А мениджира ресурсите (трие, копира, създава) В

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

Агрегация
- отношение между обекти, в което външния клас А, не отговаря за вътрешните ресурси на В
- А не се грижи за (трием, копираме, създаваме) на В
- А "използва" В

Пример:
class Singleton {
  DB provider* prov;
}

Пр: Singleton не трябва да трие DB provider!!!

Копиране на обекти

Копиращ конструктор

- приема обект от същия клас и текущия клас става 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 създаде def()

Пример:

```
struct X
{
    int a;
    char ch;
    A obj1;
    B obj2;
};
```

int a; → тетсру

A obj1; → К.К на А

B obj2; → и К.К на В

- това е автоматично генерирано от компилатора

Пример:

```
A obj1;
A obj2(obj1);
f(obj);
```

→ parse by value К.К.

void f(A obj)

Пример:

```
struct X{
    int a;
    A a;
    B b;
};
```

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

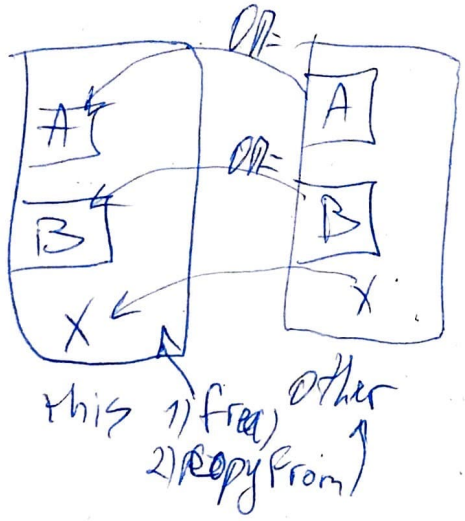
Оператор= /Оператор за присвояване

- превръща съществуващ обект в копие на друг обект
- изтрива ресурсите и след това прави копие
- автоматично генериране вика оп= на компонентите на обекти
- връща референция към настоящия обект
- дясно асоциативен оператор

Пример: извикване:

```
A obj1;
A obj2;
obj2 = obj1;
```

```
X {
  int i;
  A a;
  B b;
}
```

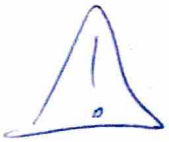


Пример a=b=c=d; дясно асоциативно

Пример: struct X {
int i;
A a;
B b;
};

X& operator=(const X& other)
{
if(this != &other){
a = other.a;
b = other.b; i = other.i;
X(const X& other): i(other.i), a(other.a),
b(other.b) {}}

Пример: A obj;
A obj2 = obj;
Тук се извиква К.К, а не ОП=, за да се оптимизира
Тук = е sugal syntax предоставен от IDE-то



A creatA(){
return A()
}

A obj = create A();
Тук се случва RVO от компилатора и се спестява К.К.
Този "трик" използван от компилатора е return value optimization

```
#pragma once
#include "Time.h"
#include "Date.h"

class Event
{
    char _name[21];
    Data _date;
    Time _startTime;
    Time _endTime;

    void setName(const char* str);
    void validateTimes();
public:
    Event();
    Event(const char* name, const Date& date,
          const Time& startTime, const Time&
                                    endTime);
    Event(const char* name, unsigned day, unsigned month, unsigned years,
          unsigned startTimeHours, unsigned startTimeMins,
          unsigned stTimeSecs, unsigned endTimeHours, unsigned
                                    endTimeMins, unsigned endTim
                                    eSecs);

    const char* getName() const;
    const Date& getDate() const;
    const Time& getStartTime() const;
    const Time& getEndTime() const;
```

Event.cpp

```
#include "Event.h"
#pragma warning(disable:4996)
Event::Event(const char* name, const Date& date,
             const Time& startTime, const Time& endTime)
    : _date(date), _startTime(startTime), _endTime(endTime)
{
    setName(name);
    validateTimes();
}

Event::Event(const char* name, unsigned day, ...)
    : _date(day, month, year),
      _startTime(startTimeHours, startTimeMins, startTimeSecs),
      _endTime(endTimeHours, endTimeMins, endTimeSecs)
{
    setName(name);
    validateTimes();
}

const char* Event::getName() const
{
    return _name;
}

const Date& getDate() const
{
    return _startTime;
}

const Time& getStartTime() const {...}
const Time& getEndTime() const {...}
```

```
void Event::setName(const char* str)
{
    if(!str || strlen(str)>20)
        return;
    else
        strcpy(_name, str);
}

void Event::validateTimes()
{
    if(compareTimes(_startTime, _endTime) <= -1)
        std::swap(_startTime, _endTime);
}

Event::Event(): Event("", 1,1,1, 0,0,0, 0,0,0){}
```