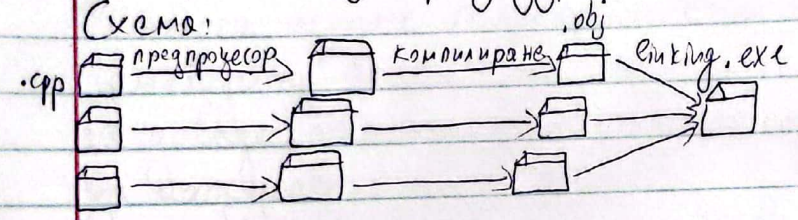
Тема 5

Разделна компилация. Копиращ конструктор и оператор =.

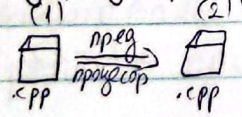
Една програма на C++ може да се разбие в множество файлове, които се компилират независимо един от друг, т.е. осъществява се разделна компилация. В резултат на компилацията Компилационният процес е следния. Разбиваме програмата на няколко файла. Тези файлове са .cpp. Първата стъпка, която се случва във всеки е предпроцесорно. Така се получават други файлове в друг формат, които са вече обработени. След това всеки един от тези файлове се компилира (минава от текстов в двоичен формат). Това вече ни дава готови обекти. Тези файлове имат разширението .obj и са независими един от друг. След това тези файлове се свързват в един .exe файл. Тази процедура се нарича Linking

Схема:



За да се възползваме максимално от разделната компилация, разделяме класовете на .h и .cpp файлове. Навсякъде, където ще ползваме класа, използваме .h файла (не се вмъкват .cpp файлове, защото така бихме имали 2 дефиниции на една и съща функция и при linking ще възникне проблем). По този начин, ако променим реализацията на някоя функция в даден клас файл, ще се прекомпилира само този файл. Накрая се изпълнява само свързването (linking) отново.

Предпроцесорни директиви - обработка на вече написан код - означава се с #



Взимаме файл (1) и след предпроцесора става на друг .cpp файл (2). Това се случва още преди програмата да се комп.

Пример кога това е от полза:
Имаме функция max, която се извиква от функцията тай определен брой пъти. Този процес е бавен, понеже всеки път в стека се пушва и попва извиканата функция max. Това може да се предотврати използвайки макроси
#define what with - замества всяко срещане на what в кода с with.
#define max(a,b) (a>b?a:b)

```
for (int i=0; i<1000; i++)
      max (4, 10);
```

} Код който извежда по-голямото от 2 число. Тази алг. е по-бърза от самото извикване на функцията.

Но макросите са доста остарели и трябва да се стремим да ги използваме минимално, защото този код не може да се дебъгва, ние нямаме достъп до този код. Ние го виждаме, но това не е кодът, който се компилира.

#include <....> - системна библиотека
#include "...." - файл, който ние сме създали.

Заедно с конструктора по подразбиране и деструктора във всеки клас се дефинират и следните член-функции:
- копиращ конструктор - конструктор, който приема обект от същия клас и създава новия обект като негово идентично копие
- Оператор = - функция/оператор, който приема обект от същия клас и променя данните на съществуващ обект от същия клас (обектът, от който е извикана функцията)

(*) Забележка: Копиращият конструктор създава нов обект, а оператор = модифицира вече съществуващ такъв.

Имплементация на копиращ конструктор

```
class A
{
  B obj1;
  C obj2;
  A(const A& other) : obj1(other.obj1), obj2(other.obj2) { }
};
```

Когато имаме композиция на обекти, при имплементирането на копиращ конструктор трябва експлицитно да извикаме копи конструкторите на обектите.
Можем да попречим на компилаторът да направи автоматичен копиращ конструктор

```
struct A
{
  A(const A& other) = delete;
};
```

Scanned with CamScanner

Имплементация на оператор=

```
class A
{
  B obj1;
  C obj2;
A& operator=(const A& other)
{
   obj1 = other.obj1; // извиква оператор=
   obj2 = other.obj2; // на член-данните
}
};
```

Оператор = връща референция на същия обект, за да е валидно гейтването.

(*)

В зависимост от кода, който сме написали, се извиква само един от тях

Пример:

```
struct Test { };
void f(Test object) { }
void g(Test& object) { }

int main()
{
  Test t; // Defaul constructor

  Test t2(t); // Copy constructor
  Test t3(t2); // Copy constructor
  t2=t3; // Operator=
  t3=t; // Operator=

  Test newTest=t; // Copy constructor, защото newTest не е създаден предварително
  f(t); // Copy constructor
  g(t); // Нищо не се извиква, тъй като подаваме по референция и не се прави копие
  Test* ptr=new Test(); // Default constructor
  delete ptr; // Destructor
} Destructor x4
```