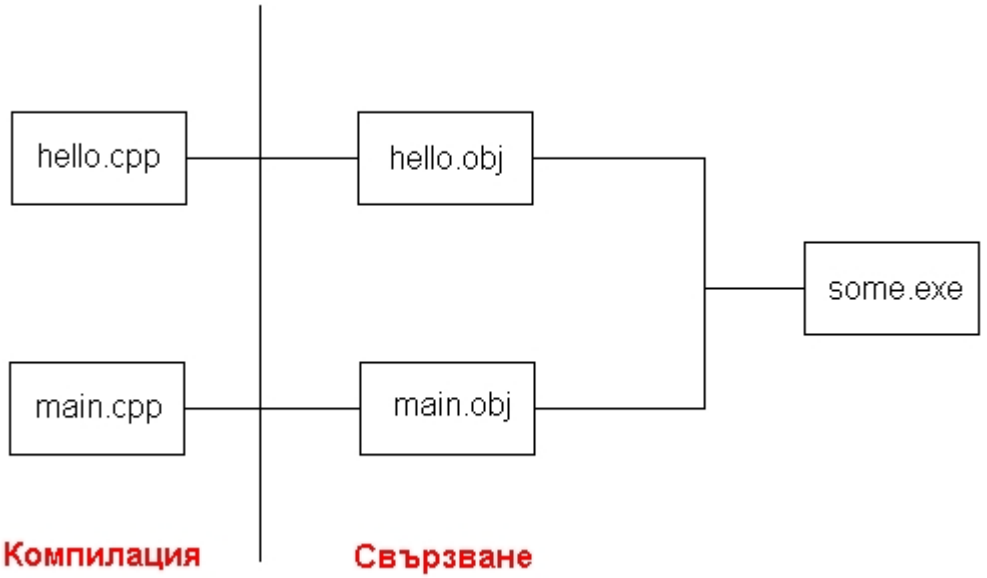
Обектно-ориентирано програмиране (записки)

- **Марина Господинова**
- **Илиан Запрянов**

Тема 05. Разделна компилация



02.cpp 01.cpp

Project7

```cpp
#include <iostream>

void f()
{
    std::cout << "02" << std::endl;
}
```

02.cpp 01.cpp*

Project7 (Global Scope)

```cpp
#include <iostream>


void f(); //обещаваме, че в процеса на свързване (Linking)
          //функцията f() ще бъде намерена (forward declaration)

          //ако функцията не бъде намерена, ще се върне грешка при Linking процеса

int main()
{
    f(); //тъй като функцията f() е дефинирана във файла 02.cpp
         //при Linking-а, 01.cpp ще намери функцията f() във 02.cpp
         //и ще я изпълни


    return 0;
}
```

02.cpp* 01.cpp*

Project7

```cpp
#include <iostream>

void f() //дефиниция на функцията f() в 02.cpp
{
    std::cout << "02" << std::endl;
}
```

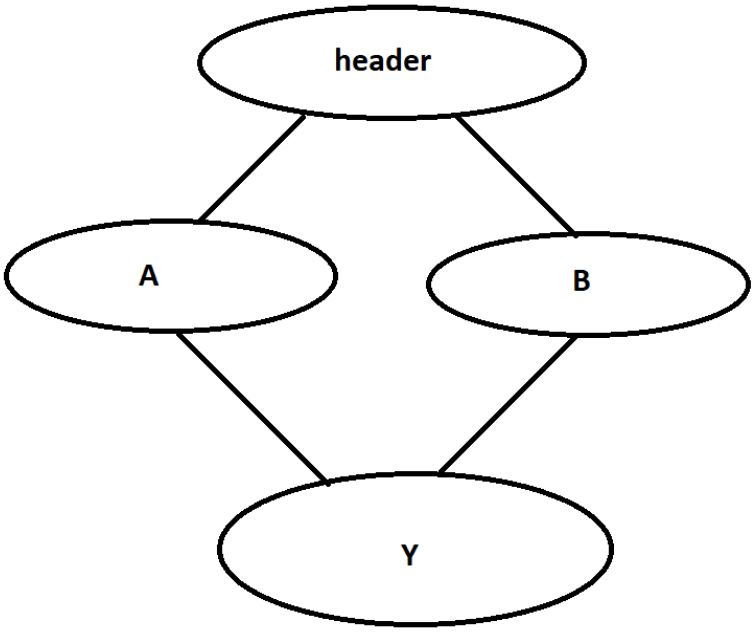
02.cpp* 01.cpp*

Project7 (Global Scope)

```cpp
#include <iostream>

void f() //дефиниция на функцията f() в 01.cpp
{
    std::cout << "01" << std::endl;
}

int main()
{
    f(); //тъй като функцията f() е дефинирана в 01.cpp и 02.cpp
         //при Linking-а, ще се намерят две функции f() с еднакви имена
         //=> конфликт на имена => няма да се компилира


    return 0;
}
```

Ще отбележим, че:

- Forward декларациите (обещанията, че функции с дадени имена ще се намерят при Linking) ще ги слагаме в .h (header) файлове и когато искаме да ги използваме ще ги include-ваме в съответния .cpp файл.
- Ако променим един .cpp файл, то тогава няма да се променят останалите, а ще се използват техните .obj файлове
- При обекти/инстанции ще пишем декларациите и обещанията за функции в .h (header) файлове, а имплементацията им в .cpp



В следния пример, нека A и B наследяват даден header файл, а Y наследява A и B. То тогава, Y include-ва 2 пъти header-а (1 път от A и 1 път от B). За да избегнем многократно включване, използваме **#pragma once**, което унищожава всяко копие на даден header и го include-ва точно веднъж.

1. Препроцесор - обработка на стрингове

Препроцесорът е първата стъпка в процеса на компилация. Той обработва всички файлове на препроцесора, които започват с #, като #include и #pragma. Примерно, когато използвате #include <iostream>, препроцесорът **замества този ред със съдържанието** на файла iostream, така че компилаторът да може да разбере и използва кода в него.

макроси - мини функции, които ни казват замести код с друг код

#define A 73 - всяко A се заменя с числото 73

```
1   #include <iostream>
2
3   //define се използа при инициализация на константи
4   //или малки функции
5
6   #define MIN(a, b) ((a) < (b) ? (a) : (b))
7   #define PI 3.14
8   int main()
9   {
10      int x = MIN(5, 3); //3
11      std::cout << PI << std::endl; //3.14
12
13
14      return 0;
15  }
```

Макросите **НЕ** заемат място в стека, по време на компилация всяко достъпване на макрос води до заместване със съответната стойност или функция.

int a = 73; **ЗАЕМА** място в стековата рамка

2. Синтактичен анализ
Тази стъпка анализира синтаксиса на изходния код, за да провери за синтактични грешки. Например, ако забравим да сложим точка и запетая (;) в края на израз, синтактичният анализ ще открие тази грешка.

a+++; **X**

3. Семантичен анализ
След като кодът е синтактично правилен, семантичният анализ проверява дали той има смисъл. Например, ако опитаме да съберем число и низ, семантичният анализ ще открие, че това е невалидно.

A obj;
obj = 73; **X**

4. Междинни оптимизации
След семантичния анализ, компилаторът може да извърши редица оптимизации на междинния код, за да направи програмата по-бърза и/или да намали нейния размер. Това може да включва премахване на излишен код или оптимизиране на цикли.

Напр.:
```
if(10 > 3) //замества се с тялото на if-а
{
   <тяло>
}
```

5. Assembly code
Програмен език от по-ниско ниво и е специфичен за типа на процесора, към който е насочена програмата. Той представлява мост между високо ниво кода и машинния код.

6. Машинен код
Асемблерният код се компилира (компилационен процес) до машинен код, който е директно изпълним от процесора. Машинният код е набор от 0 и 1. (компилирания код)

7. Линкване
След като всички изходни файлове са компилирани до машинен код, чрез Linking се комбинира този код с кода от външни файлове и генерира изпълним файл. Това е процесът на "свързване" на различни части от програмата и файловете, от които зависи, в единен изпълним файл.

8. Оптимизация - CPU-depend оптимизации
Това са оптимизации, специфични за конкретния тип процесор, към който е насочена програмата, с цел да се увеличи ефективността на програмата.

Композиция и агрегация

- взаимоотношения между обекти

Композиция

```
class A {};
class B {};
class C {};

class X
{
	A obj1;
	B obj2;
	C obj3;
};

//Композиция наричаме, когато жизненият цикъл на
// A, B, C се контролира от X, т.е.
// 1. Конструктора на X създава A, B, C
// 2. Деструктора на X унищожава A, B, C
```

```
class A {};
class B {};
class C {};

class X
{
	A* obj1;
	B& obj2;
	C obj3;
};

//Агрегация наричаме, когато
// 1. указателя или референцията сочи към обекти,
// които са самостоятелни
// 2. A, B могат да живеят извън рамките на X

//В този пример има АГРЕГАЦИЯ между
//X и A
//X и B
//
//В този пример има КОМПОЗИЦИЯ между
//X и C
```

```
class A {};
class B {};

class Y
{
	A* a;
	B& b;

	Y(A* ptr1, B* ptr2) : a(ptr1), b(*ptr2){}

	~Y() {} //не извиква деструкторите на A и B,
		//тъй като те са самостоятелни
};
```

```
class A {};
class B {};
class C {};

class Z
{
	A obj;
	B* ptr;
	C obj3;

	Z() //A(), C()
	{
		ptr = new B(); //B()

				//създаваме B (грижим се за живота на [Z])

				//макар и ptr да е пойнтър към външен ресурс,
				//това също може да се разглежда като форма на композиция,
				//тъй като [Z] управлява живота на обекта [B], към който [ptr] сочи.
				//
				//отговорността за създаването и унищожаването на обекта [B] е на класа [Z].
	} //Z()

	~Z()
	{
		delete ptr; //унищожаваме B (освобождаваме паметта; грижим се за живота на [Z])

	} //~Z(), ~B(), ~C(), ~A()
};
```

```
//нека имаме следния код

class Config
{
	int x = 4;
};

class App1
{
	Config* config = nullptr; //агрегация
public:
	App1(Config& config)
	{
		this->config = &config;
	}

	void run() { /*...*/ }
};

class App2
{
	Config* config = nullptr;  //агрегация
public:
	App2(Config& config)
	{
		this->config = &config;
	}

	void run() { /*...*/ }
};
```

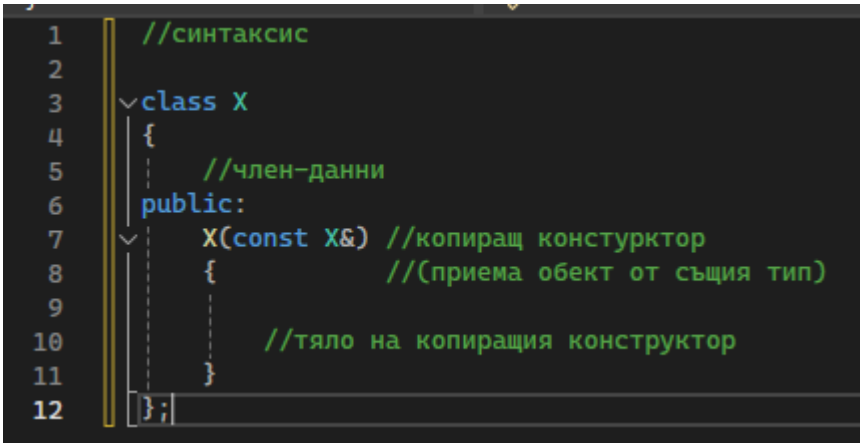
```
int main()
{
	//когато животът на [c] приключва заедно или след [a1], [a2]
	//е добър пример за агрегация, викаме конструктора, подавайки
	//външният ресурс към когото искаме да насочим пойнтър член-данната

	Config c;
	App1 a1(c);
	a1.run();

	App2 a2(c);
	a2.run();

} //в случая и двата обекта ще умрат след scope-а
```

```
int main()
{
	//когато животът на [c] приключва ПРЕДИ [a1], [a2]
	//е ЛОШ пример за агрегация

	//нека си представим, че сме разширили кода с default-ен конструктор,
	//(напомняме, че това е нужно, тъй като вече имаме наш конструктор и компилаторът
	//няма да създаде default-ен сам)

	//и имаме мутатор (setter) за член-данната [config]

	App1 a1; //викаме конструктора (default)
	{
		Config c;
		a1.set_config(c); //насочваме член-данната [config] към [c]

	} //[c] умира в края на scope-а, тоест ПРЕДИ [a1]

	//тогава нашата член-данна [config] сочи към вече освободена памет,
	//тъй като [c] вече е умряло, което очевидно е проблем

	a1.run();

}
```

Копиране на обекти

01. Копиращ конструктор

- приема обект от същия тип
- текущия става негово копие



!Ако не го създадем (разпишем), компилаторът ще създаде такъв сам!

```
class A
{
	int a;
};

class B
{
	int b;
};

class X
{
	int a;
	char ch;
	A obj; //обект
	B obj2; //обект
};
```

```
int main()
{
	X obj;
	X copy(obj); //в този пример не сме разписали
	//	^	наш копиращ конструктор, т.е. както
	//	|	вече казахме, компилаторът ще ни създаде такъв
	//	|
	//извикваме копиращия конструктор на X
	//той на свой ред извиква копиращите конструктори на A и B,
	//а на променливите от примитивен тип (в случая int и char)
	//ще извика мястото им в паметта

	//копиращият конструктор е вид КОНСТРУКТОР, които вече разгледахме,
	//тоест реда на копирането зависи  според реда на тяхното деклариране
	//в съответните класове

	//напомняме, че това е така, тъй като всяка следваща променлива/инстанция/обект
	//може да зависи от предходната променлива/инстанция/обект

	//в случая реда на деклариране е [a] [ch] [A obj] [B obj2]
	//[a] и [ch] се копират директно (тъй като са примитивни типове),
	//след което се извикват копиращия конструктор на обекта [A] (заради [obj])
	//и накрая копиращия конструктор на обекта [B] (заради [obj2])

	return 0;
}
```

```
class A
{
	int a;
};

class B
{
	int b;
};

class Y
{
	A obj1; //обект
	B obj2; //обект

public:
	//предният пример може да ни позволи да
	//направим следния извод:

	//декларацията ни на член-данните е A -> B
	//=> след извикването на копиращия конструктор на Y
	//редът на копиране ще е A -> B

	Y(const Y& other) // при копиране сме извикали копиращия на Y
	{
		//това се случва всъщност
		//obj1(other.obj1); -> извикваме копиращия на A
		//obj2(other.obj2); -> извикваме копиращия на B
	}
};
```

```
int main()
{
	//продължавайки горния пример:

	//викане на копиращ конструктор

	A obj; //викане на DEFAULT конструктор
	A obj2(obj); //викане на КОПИРАЩИЯ конструктор, подавайки обекта [obj]

	return 0;
}
```

```
int main()
{
	A obj;

	f(obj); //функцията [f] приема копие от тип [A]
		//тоест [a] ще копира [obj] (вика се копиращия конструктор)

	g(obj); //функцията [g] приема РЕФЕРЕНЦИЯ към тип [A]
		//тоест [a] е референция на [obj]

		//копиращият конструктор НЯМА да се извика,
		//тъй като правим референция към [obj], а не го копираме

	return 0;
}
```

02. Оператор = (оператор за присвояване)

- приема обект от същия тип и текущия става негово копие
- текущият обект е съществувал преди това

```
A obj1;
A obj2;

obj1 = obj2; //obj1 вече Е СЪЩЕСТВУВАЛ
		//всички данни на obj2 се копират в obj1
```

Разликата между копиращия конструктор и оператор= е:

копиращия конструктор

01. копира данните

оператор=

01. изчиства текущите данни

02. копира данните

!Ако не създадем оператор= (разпишем), компилаторът ще създаде такъв сам!

```
class A
{
	int a;
};

class B
{
	int b;
};

class X
{
	A obj1; //обект
	B obj2; //обект
};


int main()
{
	X obj1;
	X obj2;

	obj2 = obj1; //извикване на оператор= (в случая default-ния такъв)

	//подобно на default-ния копиращ конструктор, тук се
	//извиква първо оператор= на [X], след което и тези на [A] и [B]
	//(след [X] викането на операторите= зависи от реда на декларация на
	//член-данните (отново за примитивни типове просто се копират данните))


	return 0;
}
```

```
class A
{
	int a;
};

class B
{
	int b;
};

class X
{
	A obj1; //обект
	B obj2; //обект
public:
	X& operator=(const X& other) //забелязваме, че [operator=] връща РЕФЕРЕНЦИЯ
	{			//това е така, за да можем да използваме [operator=] верижно

				//това нямаше да бъде възможно ако [operator=] беше [void],
				//тъй като нямаше да връщаме нищо:

				//obj3 = obj2 = obj1; щеше да доведе до компилационна грешка,
				//защото obj2 = obj1 няма да върне стойност, която може да се присвои на obj3

				//а връщането по референция, а не по копие, за да спестим ненужно копиране
				//(с връщане по копие [operator=] все още ще работи, но по-неефективно)

		obj1 = other.obj1;
		obj2 = other.obj2;

		return *this;
	}
};
```

```
int main()
{
	X obj1;
	X obj2;

	obj2 = obj1; //извикване на оператор=

	X obj3;
	obj3 = obj2 = obj1; //верижно ползване на operator=


	return 0;
}
```

```
int main()
{
	X obj1;
	X obj2 = obj1; //обръщаме внимание, че за да се извика operator=
	               //има изискване и двата обекта да съществуват,
	               //тъй като [obj2] не съществува, а го създаваме в момента,
	               //това ще извика копиращия конструктор, където
	               //[obj2] ще се създаде и ще му бъдат зададени стойностите на [obj1]

	return 0;
}
```

```
int main()
{
	X obj1;
	X obj2;

	obj1 = obj2; //тъй като [obj1] вече съществува, ще се извика [operator=],
	             //който по дефиниция ще изпълни следните две стъпки
	             //01. ще изтрие данните на [obj1]
	             //02. ще копира в [obj1] данните на [obj2]


	return 0;
}
```

```
int main()
{
	X obj1;

	obj1 = obj1; //тъй като [obj1] вече съществува, ще се извика [operator=],
		     //който по дефиниция ще изпълни следните две стъпки
		     //01. ще изтрие данните на [obj1] (левият обект)
		     //02. ще копира в [obj1] (левият обект) данните на [obj1] (десният обект)

	//това обаче води до очевиден проблем,
	//тъй като първата стъпка е да се изтрият данните на [obj1] (левият обект) =>
	//в [obj1] (левият обект) не можем да присвоим данните на [obj1] (десният обект),
	//тъй като обектите са един и същ, и с триенето на данните на левия са изтрити и тези на десния,
	//(тоест на левия обект искаме да присвоим изтрити данни)


	return 0;
}
```

```
class X
{
	A obj1; //обект
	B obj2; //обект
public:

	X& operator=(const X& other)
	{
		if (this != &other) //проверяваме дали обектите са различни,
		{                   //ако са, няма да има проблем при стъпка 01

			obj1 = other.obj1;
			obj2 = other.obj2;

		}

		return *this; //ако НЕ СА различни, просто ще върнем непроменения наш обект
			      //(няма да влезем в if-а)

			      //ако СА различни, ще върнем променения наш обект
	}
};


int main()
{
	X obj1;

	obj1 = obj1; //вече сме решили проблема

	return 0;
}
```

RVO/NRVO

```
//RVO и NRVO се прилагат автоматично от компилатора
//(целят да подобят ефективността на нашия код)

//RVO - return value optimization
//спестява копиращия конструктор
//(когато обекта на връщане НЯМА име)
A createA()
{
    return A(); //извиква се default-ният конструктор на A
}

A obj = createA() //очаква се да се извика копиращия конструктор,
                  //но това не се случва заради RVO оптимизацията,
                  //тоест влизаме във функцията, викаме default-ния
                  //конструктор на A и благодарение на RVO оптимизацията
                  //не се вика копиращия конструктор, въпреки това в [obj]
                  //се запазват данните

//NRVO - named return value optimization
//спестява копиращия конструктор
//(когато обекта на връщане ИМА име)
A createA()
{
    A obj; // извиква се default-ният конструктор на A

    return obj;
}

A obj = createA() //очаква се да се извика копиращия конструктор,
                  //но това не се случва заради NRVO оптимизацията,
                  //тоест влизаме във функцията, викаме default-ния
                  //конструктор на A и благодарение на NRVO оптимизацията
                  //не се вика копиращия конструктор, въпреки това в [obj]
                  //се запазват данните
```