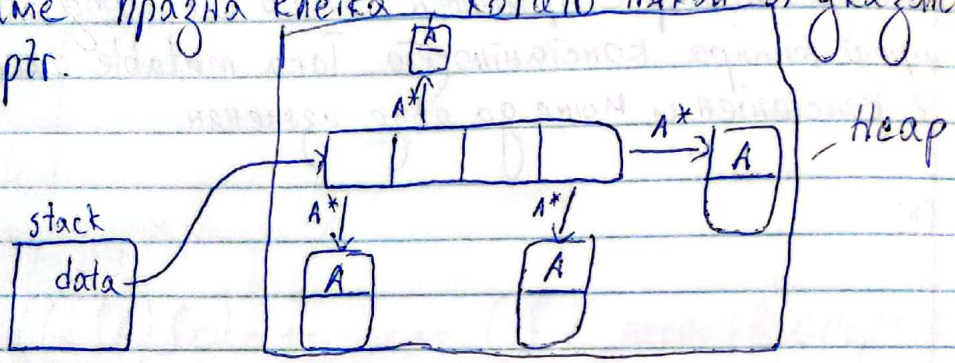
Тема 8

Масиви от указатели към обекти. Изключения. Обработка на изключения. Проблеми при работа с потоци. Move семантики - lvalue, rvalue, move конструктор / оператор =, std::move.

Когато искаме да пазим колекция от обекти, можем да го направим чрез масиви от указатели към обектите. Това ни позволява бързо разместване на обектите в колекцията, като не копираме целите класове, а само разместваме указателите. Освен това не се изисква default-ен конструктор за класа, в който са обектите и свободната е възможно да имаме "празна клетка" която няква от указателите има стойност nullptr.



Пример:

```
class someCollection
{
  A** data;
  // other things;
};
```

Изключенията са възможност да сигнализираме, ако се случи нещо неочаквано във функция. Грешките се "хвърлят" с ключовата дума throw и "хвърляме" някакъв обект - throw <object>
Грешките трябва да се обработват без да се губи информация за тях и трябва да дават максимално добра информация за случилата се грешка. За целта се използва следния синтаксис

```
try
{
  // код, в който може да - Ако тук възникне някаква грешка, ние можем
} възникне грешка      да я уловим с функцията catch.
```

Можем да правим различна catch клаузи в зависимост от това, което искаме да "хванем":

```
catch(int)
{
  // обработка на int
} вид грешка

catch(bool)
{
  // обработка на int и bool
}
```

Вместо да "хвърляме" int, bool и т.н. можем да "хвърляме" следните класове: std::exception (използва се често при наследяване), std::logic_error, std::invalid_argument и други.

Има варианти за използване следния синтаксис:
catch(// some error class // ако "хване" тук проблеми, се изпълнява кода, а ако не:
catch(// other error class) // пробва тук, а ако не:
catch(// other error class) // пробва тук, а ако не:
....
catch(...) // ако грешката не се "хване" в предните, идва тук.
Тази структура се нарича последователно "catch-хване" и наподобява if-else структурата. (catch(...) = else statement)

Забележка: Не е добра идея да се използва catch(...), защото се губи конкретиката на грешката.

Особености:

```
struct A
{
 ~A()
 {
  std::cout << "~A" << std::endl;
 }
};

struct B
{
 ~B()
 {
  std::cout << "~B()" << std::endl;
 }
};

void g()
{
 B obj2;
 throw 42; -> извиква деструктора на obj2 // "~B()"
}
void f()
{
 A obj;
 g(); -> извиква деструктора на obj // "~A()"
 A obj2; -> този обект не се създава
}
```

```
int main()
{
    f(...);
}
```

по принцип

ако се обработи грешката

- тази функция ще изкара грешка, но ~~преди това~~ ще отпечата "B(1)" и "~A()"

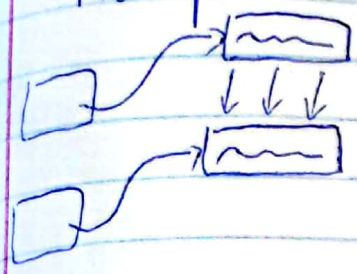
Когато стартира програмата в стека се трупат една над друга функциите g() - тук имаме "хвърляне на обект 42; затова дотърси надолу кой е първия f() catch поредя. Виждаме, че грешката не се обработва в никоя от main() функциите, грешката търси catch в g(), f() и main() и тъй като не се catch-ва никъде, тя минава навсякъде в обратен ред, като по пътя извиква деструкторите на всички създадени вече локални обекти

Какво се случва, ако "хвърлим" грешка в деструктор?
Тъй като можем да поддържаме само една хвърлена грешка, не бива да "хвърляме" грешка в деструктора.
Силно препоръчително е да извикваме .close() на потоците експлицитно, тъй като .close() може да "хвърли" грешка и не трябва да позволяваме на самия деструктор да извика функцията .close(), защото може деструкторът да бъде извикан в следствие на друга "хвърлена" грешка и ще имаме повече от една "хвърлени" грешки наведнъж, която не можем да обработим. За да избегнем тези сценарии, винаги се стремим да затваряме потоците ръчно:

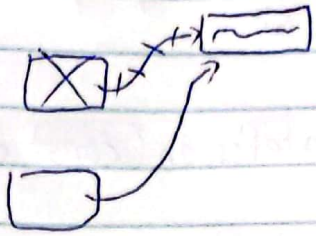
```
try
{
  file.close();
}
catch (ios::fail);
```

Move семантики (местене на обекти) - използват се, когато искаме да "преместим" стойността на даден обект на друг обект. Прави се в случай, когато сме сигурни, че първо-началният обект няма да се използва занапред

Копиране:



Move семантика:



RVO - return value optimization. Понякога компилаторът е достатъчно "умен", за да прецени кога да "премести" стойността, вместо да прави копие, но ние можем експлицитно да му покажем, че искаме да "преместим" стойността използвайки функцията std::move(lvalue).

lvalue-обект, за който има заделена памет
rvalue-всичко което не е lvalue

```
int a;
a = 7
```

lvalue rvalue

операции:

lvalue:
a+=b
връща променлива
-=, /=, *=

rvalue
a+b
връща стойност
*, -, /

Пример:

```
void f(int& n)
{
 std::cout<<n<<std::endl;
}
int main()
{
 int number = 10;
 f(number); //валидно
 f(10); //компилационна грешка, защото 10 няма адрес в паметта
 g(number); //невалидно, приема само rvalue
 g(10); //валидно
```

```
void g(int&& n)
{
 std::cout<<n<<std::endl;
}
```

Да разгледаме следния програмен фрагмент:

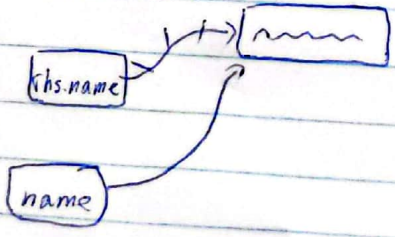
```
#include "Person.h"

Person createPerson(const char* name, int age)
{
    Person p(name, age);
    return p; - копие
}

int main()
{
  Person p2("Peter", 18);
  p2 = createPerson("Ivan", 23);
}
```

Тук се извикват следните функции на Person:
1. Конструктор на Person (за p2)
2. Конструктор на Person (за p)
3. Копиращ конструктор (за return p)
4. Деструктор (в края на функцията createPerson())
5. Оператор= (за p2 = createPerson(..))
6. Деструктор (в края на main функцията)

Проблемът тук е, че правим излишно копие. Обектът, създаден в createPerson, се копира 2 пъти, докато се присвои на p2

За да избегнем излишните копия, можем да "преместим" данните на обектът, който се създава във функцията в данните на p2.

```
// Move constructor
Person::Person(Person&& rhs)
{
  name = rhs.name;
  age = rhs.age;
  rhs.name = nullptr;
}
```



```
//Move operator=
Person& Person::operator=(Person&& rhs)
{
  if (this!=&other)
    {
      free();
      name=rhs.name;
      rhs.name=nullptr;
      age=rhs.age;
    }
  return *this;
}
```

Важно е да ее обърне внимание, че тук данните не се копират, а се местят. Т.е. приемаме, че rhs няма да ще използва след изпълнението на функцията. Сега при изпълнението на първоначалния код:

1. Конструктор за p2
2. Конструктор за p
3. Move конструктор за return p
4. Деструктор (в края на функцията createPerson)
5. Move оператор= (за p2=createPerson..)
6. Деструктор (в края на main функцията)

Тоест тук си спестихме две копия на динамичната памет в Person.