C++ learning documentation

# УП

## Headers

* Header-ите се използват като пакети, в които може да си разпределим кода, може директно да дефинираме методи и класове там. Най-често класове се дефинират там и после се имплементират в .cpp файл.

## Strings

* string s, s.resize(), s[int]

## Указатели, рефенреции

### Дефиниция

* Указателя е тип променлива, която пази референцията към някаква променлива.
* Референцията е адреса в паметта, в който е запаметен дадена променлива.

### Примери

* int var a = 100;
* cout << &a; // референция Резултат: 0x6afefc
* int \*a1 = &a; // тук а1 е указател
* cout << a1; //Резултат: 0x6afefc
* a1 == &a; // Резултат: true;
* със \*<името на указателя> достъпваме стойността на адреса, на който сочи указателя;
* cout << \*a1; // Резултат: 100;
* \*a1 = 1234; // Резултат: променя стойността на адреса, на който сочи указателя;
* cout << a; // Резултат: 1234

### Константни указатели и променливи

#### Указатели към константни променливи

* Не може да се промени стойността в адреса на променливата
* Може да се промени стойността на указателя (да сочи на друго място)
* Пример:
* const int var1 = 100; const int var2 = 200;
* const int \*varP = &var1;
* \*varP = 200; // НЕ Е ПОЗВОЛЕНО (Смяна на стойността)
* varP = &var2; //ПОЗВОЛЕНО (Смяна на адреса)

#### Константен указател към променлива

* Не може да се промени стойността на указателя (да сочи на друго място)
* Може да се промени стойността на променливата
* Пример:
* int var1 = 100; int var2 = 200;
* int \* const varP = &var1;
* varP = &var2; // НЕ Е ПОЗВОЛЕНО (Смяна на адреса)
* \*varP = 300; // ПОЗВОЛЕНО (Смяна на стойността на var1)

#### Константен указател към константна променлива

* Аналогично
* Не може да се промени стойността на променливата
* Не може да се промени адреса на към който сочи указателя

## Работа с масиви

* Масивите се подават на функции чрез указатели. Не може да вземем дължината на масив в функция.
* Масивите нямат пряка функция за големина.
* Можем да променим съдържанието на масива във функцията
* Не можем да променим указателя на масива към друг масив, създаден във функцията, защото тази референция се унищожава след извикването на функцията.
* 2D масив може да се подаде като параметър на функция по 3 различни начина, в зависимост как е дефиниран.
* Int a[][] -> (int a[][])
* Int \*array[] -> (int \*a[])
* Int \*\*array -> (int \*\*a)

# ООП

## Stack vs Heap

* Stack -статична памет, заделя се докато се компилира програмата и не може да се промени run-time
* Heap – динамична памет, която може да се променя според run-time-а на програмата, dynamic memory allocation. Примерно ако имаме променлива и искаме да заделим масив с толкова елемента в паметта, ще ни трябва heap, защото не може да се изчисли колко памет ще трябва да се задели в stack-a докато се компилира програмата, защото не знаем каква ще е стойността на run-time променливата.
* Objects in the stack
* Objects in the heap
* When pointers come into play – ползваме pointers, когато искаме да достъпим нещо в heap-a.
* Stack memory-то наподобява структурата от данни Stack, но Heap memory-то не наподобява структурата от данни Heap.
* new – заделя памет в heap-a ПРИМЕР: int \*p; p = new int; \*p = 10; p = new int[20];
* delete – трие съдържанието, на към което сочи pointer-a ПРИМЕР: delete p; delete[] p;

## :: оператор

* В C++ може да дефинираме функционалността на даден метод, извън класа с оператор ::
* spaceholder
* class Human {   
  public:   
  string name;   
  void hello();  
  } ;   
  void Human :: hello () {   
  cout << Human :: name << “said hello”;  
  }
* Не може да задаваме default стойности на променливи по същия начин освен ако не са статични.

## Access modifiers

* private
* public

## Constructors

* специална функция, без return type със същото име като на класа
* default конструктора се извиква автоматично, когато инициализираме инстанция на класа в Stack-a
* default конструктора не се извиква автоматично, когато инициализираме инстанция на класа с различен конструктор;
* Когато инициализираме инстанция на класа в Heap-a може да го направим по следния начин: Person \*me = new Person; това извиква автоматично default конструктора.
* Ако не сме дефинирали конструктори за клас, то може да ползваме default конструктора, но ако дефинираме някакъв конструктор за даден клас, то не може да ползваме default конструктора освен ако не го дефинираме.
* Може да задаваме default стойности както на функции така и на конструктори по следния начин:
* void sayHello(string name = “NO NAME”, int age = -1) {  
  cout << “my name is: ” << name << “ and my age is: ” << age;  
  }
* sayHello(); // my name is: NO NAME and my age is -1
* sayHello(“elizar”); // my name is: elizar and my age is -1
* sayHello(10); // my name is: NO NAME and my age is: 10
* sayHello(“elizar”, 10) // my name is: elizar and my age is: 10

## Destructors

* Special member function of a class that is executed when an object of this class goes out of scope or delete expression is applied to a pointer to an object of that class
* ~Human() {}
* Деструкторите не приемат параметри
* Използват се за изчистването на ресурсите, в heap-a използвани от даден pointer

## Static

* Държи се като в Java
* Трябва да дефинираме статичните неща по следния начин:
* Class Human {  
  public:   
  static int total\_humans;  
  void getTotalHumans(){  
  cout << total\_humans;  
  }  
  }  
    
  int Human::total\_humans = 0;  
    
  int main(){cout << Human::total\_humans}
* Може да ползваме статичен брояч за да видим колко обекта има създадени от даден клас, увеличаваме брояча в конструктора и намаляваме брояча в деструктора

## Friend functions

* Ако трябва да достъпим елементите на клас, които са private извън класа ползваме приятелски функции.
* Може да имаме приятелски функции и приятелски класове.
* Във C++ може да предефинираме оператори +, -, \*, /

## Наследяване

* Може да ограничим наследяването чрез специфичен access modifier
* Public -> public and protected act as public and protected
* Protected -> public and protected act as protected
* Private -> public and private act as private
* В cpp може да пренапишем методи в наследен клас, това води до полиморфизъм
* Може да достъпим пренаписаните методи в родителския клас
* За да имаме полиморфизъм на метод, той трябва да е виртуален.

## This

* this e pointer
* приятелски функции и класове нямат достъп до this

## Abstract

* не обозначаваме абстрактните класове
* абстрактен клас е клас със поне една pure virtual function
* The diamond problem
* Virtual наследяване

# СДА

## Колекции

### vector