Класове

Обектно ориентирано програмиране - семинар 2022/2023

Класове - идея, видимост

Възникване на класовете

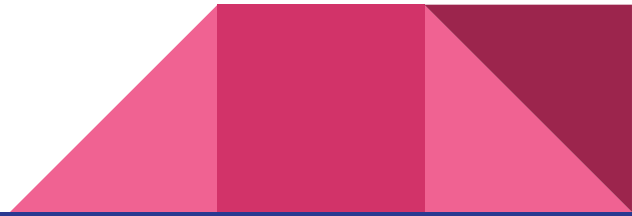
- 1972 г. - Излиза езикът C, който съдържа структури (struct). Това обаче е недостатъчно за едно по-голямо приложение.
- 1979 г. - Създава се модификация на C (C with classes), която добавя класовете, но поради недостатъчни спецификации и множеството ограничения, този проект претърпява провал
- 1982 г. - Създателя на C with classes, започва целия проект наново, като добавя множество модификации, така че езикът от процедурен да се превърне в обектно ориентиран - така се появява C++

Какво е клас?

Основната функционална характеристика на C ++ е класът (затова и езикът се нарича обектно-ориентиран). Класът е съставен тип данни, дефиниран от потребителя, който има за цел да представя определена концепция в кода на дадена програма (кола, къща, човек и тн). Всеки път, когато нашият обектно-ориентиран дизайн има полезна концепция, идея, същност и т.н., трябва да я представим като клас в програмата, така че идеята да е там в кода, а не само в главата ни. Програма, изградена от добре подбран набор от класове е много по-лесна за разбиране, отколкото този, който изгражда всичко директно с примитивните типове.

Основни принципи на ООП

- Енкапсулация
- Наследяване
- Абстракция
- Полиморфизъм

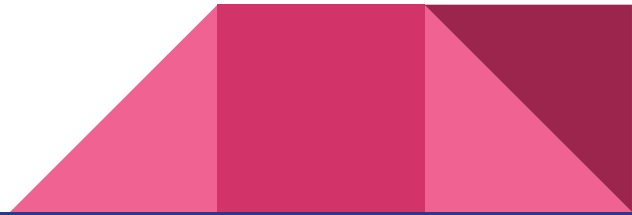


Енкапсулация

Целта на енкапсулацията е да осигури че ‘чувствителните’ данни са скрити от потребителя. По тази причина се използва ключовата дума private за тези полета. Когато искаме да можем все пак да достъпим техните стойности, използваме get и set методите им, които са публични.

private - достъп само в зоната на класа

public - достъп в целия код



Декларация на клас

class <Име на класа> {

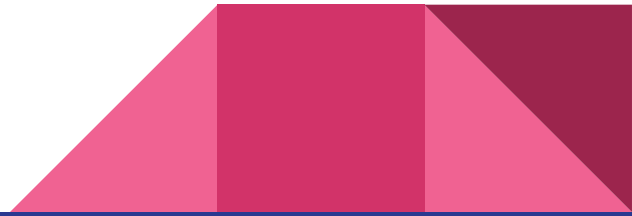
private:

… //всички полета и методи на класа, които са скрити от потребителя

public:

… //всички полета и методи на класа, които са достъпни от потребителя

};

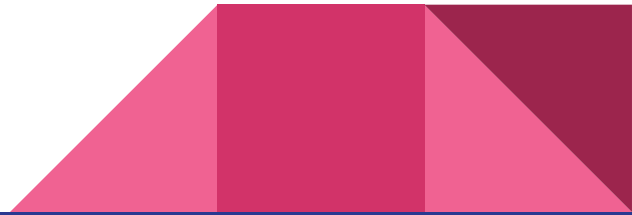


Полета и методи

Полета и методи

Полета - характеристики на класа, които дават неговото по-пълно описание. Те трябва да бъдат, не прекалено, но достатъчно описателни, така че да може да описват тяхното състояние, по разбираем и удобен за използване от потребителя начин

Методи - действия които се извършват върху полетата, така че състоянието на обектите от този клас да бъдат



Пример

Полета

```
class Employee {
  private:
    // Private attribute
    int salary;
```

Методи

```
public:
  // Setter
  void setSalary(int s) {
    salary = s;
  }
  // Getter
  int getSalary() {
    return salary;
  }
```

Вътрешни и външни методи

```
class MyClass {        // The class
  public:              // Access specifier
    void myMethod() {  // Method/function defined inside the class
      cout << "Hello World!";
    }
};
```

<- Вътрешен метод

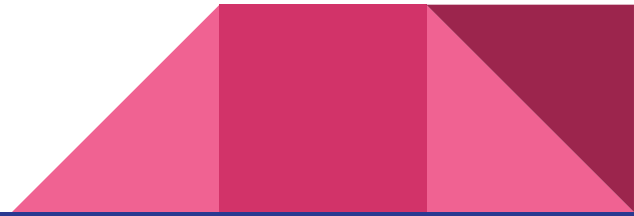
Външен метод ->

```
class MyClass {        // The class
  public:              // Access specifier
    void myMethod();   // Method/function declaration
};

// Method/function definition outside the class
void MyClass::myMethod() {
  cout << "Hello World!";
}
```

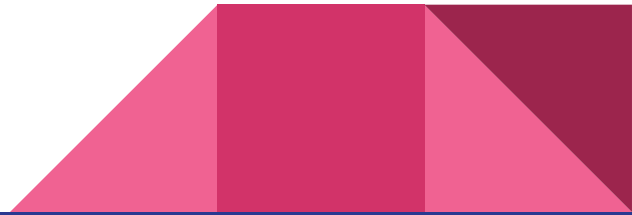
Класове и обекти - каква е разликата?

Класът е шаблон за създаване на обекти - описва как ще изглежда обект от този клас. Обектът е инстанция на класа. При създаване на обекти от даден клас, те наследяват всички полета и методи от този клас.

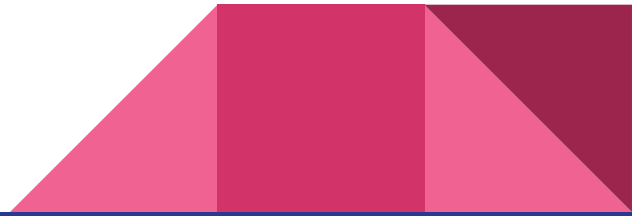


Указателя this

Важно уточнение за класовете е, че за всеки обект имаме различна инстанция на полетата и една споделена инстанция на методите. Проблемът, който се получава е че тъй като методите са достъпвани от различни обекти, то има случай в които подаваме като параметри други обекти от същия клас, или полета със същите имена. Тогава за да можем да отличим полетата на текущата инстанция (инстанцията която вика метода) използваме указателя this (той е от тип &<име на класа>).



Жизнен цикъл. Конструктори и деструктори



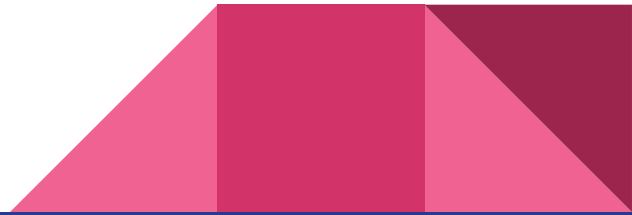
Конструктор

Метод на класа който се извиква при създаване на обект и се използва инициализация на неговите полета.

Декларация: <име на класа>() - конструктора няма тип и за име използва името на класа. Когато няма параметри се нарича конструктор по подразбиране.

Ако не напишем конструктор по подразбиране, то се създава такъв автоматично, но има празно тяло.

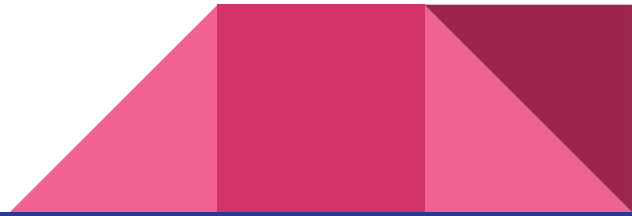
Можем да дефинираме много видове конструктори.



Деструктор

При използване на динамична памет при създаване на обект е нужно да се освободи, когато обекта вече не се използва. Тук идва ролята на деструктора

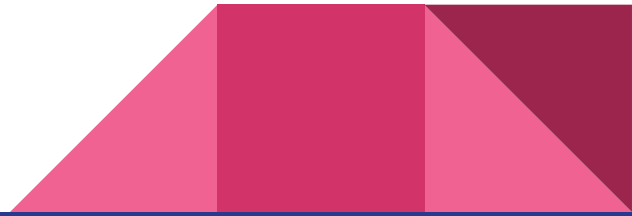
Декларация: ~<име на класа>()



Управление на динамична памет

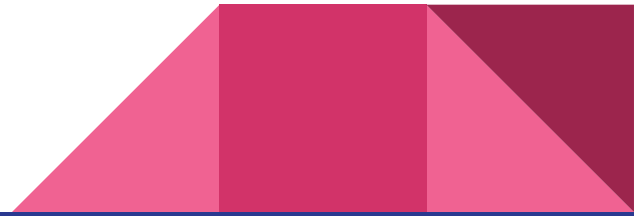
Когато имаме динамична памет като поле, т.е поле от тип указател е нужно да създадем т. нар. голяма четворка. Тя съдържа:

- Конструктор по подразбиране
- Конструктор за копиране
- Предефиниране на оператор =
- Деструктор



Задача

Да се напише клас Employee с име, години и работен номер. Името и работният номер да бъдат от тип char*. Да се реализират всички методи осигуряващи енкапсулация и управление на паметта.



Изключения (exceptions)

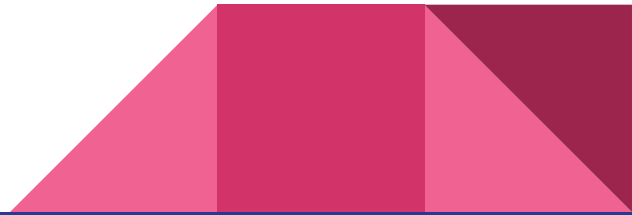
Когато има грешка в нашата програма, C++ нормално спира и извежда съобщение за грешка. Този процес се нарича хвърляне на изключение(throw exception). Тук имаме 3 ключови думи:

try - позволява да дефинираме блок от код, който да бъде тестван за грешки по време на изпълнение

catch - позволява да дефинираме блок от код, който да бъде изпълнен ако се получи дадената грешка

throw - хвърля грешката когато е открита

try и **catch** винаги вървят заедно



Изключения - пример

```
try {
  int age = 15;
  if (age >= 18) {
    cout << "Access granted - you are old enough.";
  } else {
    throw (age);
  }
}
catch (int myNum) {
  cout << "Access denied - You must be at least 18 years old.\n";
  cout << "Age is: " << myNum;
}
```

Универсална обработка на изключения (...)

```
try {
  int age = 15;
  if (age >= 18) {
    cout << "Access granted - you are old enough.";
  } else {
    throw 505;
  }
}
catch (...) {
  cout << "Access denied - You must be at least 18 years old.\n";
}
```

Влагане на класове

Association	Aggregation	Composition
Class A **uses** Class B.	Class A is **owns** Class B.	Class A **contains** Class B.
Example: • Employee uses BusService for transportation. • Client-Server model. • Computer uses keyboard as input device.	**Example:** • Manager has N Employees for a project. • Team has Players.	**Example:** • Order consists of LineItems. • Body consists of Arm, Head, Legs. • BankAccount consists of Balance and TransactionHistory.
An association is **used when** one object wants another object to perform a service for it. Eg. Computer uses keyboard as input device.	An aggregation is **used when** life of object is independent of container object But still container object owns the aggregated object. Eg. Team has players. If team dissolve, Player will still exists.	A composition is **used where** each part may belong to only one whole at a time. Eg. A line item is part of an order so A line item cannot exist without an order.

Composition

```
sum of 20 and 100 = 120
sum of 20 and 5 = 25
```

```
class X
{
    private:
    int d;
    public:
    void set_value(int k)
    {
        d=k;

    }

    void show_sum(int n)
    {
        cout<<"sum of "<<d<<" and "<<n<<" = "<<d+n<<endl;

    }


};
class Y
{
    public:
    X a;
    void print_result()
    {
        a.show_sum(5);

    }

};

int main()
{
  Y b;
  b.a.set_value(20);
  b.a.show_sum(100);
  b.print_result();

}
```

Aggregation

```
class Teacher
{
private:
  std::string m_name{};

public:
  Teacher(const std::string& name)
      : m_name{ name }
  {
  }

  const std::string& getName() const { return m_name; }
};

class Department
{
private:
  const Teacher& m_teacher; // This dept holds only one teacher for simplicity, but it could hold many
teachers

public:
  Department(const Teacher& teacher)
      : m_teacher{ teacher }
  {
  }
};
```

Статични полета и методи

Приятелски функции

```
class ABC;

class XYZ {
    int x;

public:
    void set_data(int a)
    {
      x = a;
    }

    friend void max(XYZ, ABC);
};
```

```
class ABC {
    int y;

public:
    void set_data(int a)
    {
      y = a;
    }

    friend void max(XYZ, ABC);
};
```

```
void max(XYZ t1, ABC t2)
{
    if (t1.x > t2.y)
        cout << t1.x;
    else
        cout << t2.y;
}
```

Singleton pattern

Singleton in C++ / Design Patterns (refactoring.guru)