ООП специфики. Класове.

доц. д-р Нора Ангелова

ООП

Обектно-ориентираното програмиране (ООП) е парадигма в компютърното програмиране, при която една програмна система се моделира като набор от обекти, които взаимодействат помежду си.

ООП

- Принципи на ООП
  - Абстракция - Abstraction
  - Капсулация – Encapsulation
  - Наследяване – Inheritance
  - Полиморфизъм – Polymorphism

КЛАСОВЕ

В езика C++ има стандартен набор от типове данни като int, double, float, char, string и др. Този набор може да бъде разширен чрез дефинирането на класове.

Дефинирането на клас(по подобие на структурите) въвежда нов тип, който може да бъде интегриран в езика.

Класовете и структурите са в основата на ООП.

КЛАСОВЕ

- Класовете са подобни на записите (структурите)
- Всички неща, които знаем за структури важат и за класовете
- Какво липсва
  - Класовете имат допълнителни ограничения по отношение на правата за достъп

КЛАСОВЕ

- Защо искаме да използваме класове и структури
- Нива на абстракция и цел

КЛАСОВЕ (НИВА НА АБСТРАКЦИЯ)

КЛАСОВЕ

- Декларация на клас

```
class <име_на_клас>;
```

Пример:

```
class Point2D;
```

КЛАСОВЕ

- Дефиниция на клас

```
class {<име_на_клас>}опц {
  {<спецификатор_за_достъп>:}опц
  <член_данни>;|{<член_данни>;}опц

  {<спецификатор_за_достъп>:}опц
  <член_функции>;|{<член_функции>;}опц
} [<деф_на_обекти>];
```

Пример:

```
class Point2D {
  // …
};
```

КЛАСОВЕ

- Дефиниция на клас

```
class <име_на_клас> {
  {<спецификатор_за_достъп>:}опц
  <член_данни>;|{<член_данни>;}опц

  {<спецификатор_за_достъп>:}опц
  <член_функции>;|{<член_функции>;}опц
};
```

- Спецификаторите за достъп мога да бъдат:
  - `public`
  - `private`
  - `protected`

* *Спецификаторите за достъп могат да се използват и в структури. Разликата между класове и структури е в спецификатора по подразбиране*

Класове

- Спецификатори за достъп
  - Могат да бъдат пропускани.
  - По подразбиране спецификаторът за достъп в записа (структурата) е public.
  - По подразбиране спецификаторът за достъп в класовете е private.
  - Областта на един спецификатор за достъп започва от спецификатора и продължава до следващия спецификатор или до края на декларацията на класа.

КЛАСОВЕ

- Дефиниция на клас

```
class <име_на_клас> {
  {<спецификатор_за_достъп>:}опц
  <член_данни>;|{<член_данни>;}опц

  {<спецификатор_за_достъп>:}опц
  {<член_функции>;|{<член_функции>;}опц}опц
};
```

Пример:

```
class Point2D {
  private:
  //...
  public:
  //...
};
```

КЛАСОВЕ

- Дефиниция на клас

```
class <име_на_клас> {
  {<спецификатор_за_достъп>:}опц
  <член_данни>;|{<член_данни>;}опц

  {<спецификатор_за_достъп>:}опц
  {<член_функции>;|{<член_функции>;}опц}опц
};
```

- Член-данни

```
<член_данни> ::= <име_на_тип> <име>{, <име2>}опц;
```

КЛАСОВЕ

- Дефиниция на клас

```
class <име_на_клас> {
  {<спецификатор_за_достъп>:}опц
  <член_данни>;|{<член_данни>;}опц

  {<спецификатор_за_достъп>:}опц
  {<член_функции>;|{<член_функции>;}опц}опц
};
```

- Член-данни

Пример:

```
class Point2D {
  private:
    double x;
    double y;
  public:
    // …
};
```

КЛАСОВЕ

- Дефиниция на клас

```
class <име_на_клас> {
  {<спецификатор_за_достъп>:}опц
  <член_данни>;|{<член_данни>;}опц

  {<спецификатор_за_достъп>:}опц
  {<член_функции>;|{<член_функции>;}опц}опц
};
```

- Член-функции

```
<декларация_член_функции> ::=
 <име_на_тип> <име> ({<списък_формални_параметри>}опц);
```

КЛАСОВЕ

- Дефиниция на клас

```
class <име_на_клас> {
  {<спецификатор_за_достъп>:}опц
  <член_данни>;|{<член_данни>;}опц

  {<спецификатор_за_достъп>:}опц
  {<член_функции>;|{<член_функции>;}опц}опц
};
```

- Член-функции
  - Извикват се с обект на класа (имат неявен параметър this)
  - Работят с член-данните на класа за този обект

КЛАСОВЕ

- Дефиниция на клас

```
class <име_на_клас> {
  {<спецификатор_за_достъп>:}опц
  <член_данни>;|{<член_данни>;}опц

  {<спецификатор_за_достъп>:}опц
  {<член_функции>;|{<член_функции>;}опц}опц
};
```

- Член-функции

Пример:

```
class Point2D {
  private:
    double x;
    double y;
  public:
    void print() const;
};
```

КЛАСОВЕ

- Дефиниция на клас

```
class <име_на_клас> {
  {<спецификатор_за_достъп>:}опц
  <член_данни>;|{<член_данни>;}опц

  {<спецификатор_за_достъп>:}опц
  {<член_функции>;|{<член_функции>;}опц}опц
};
```

- Член-функции

```
<дефиниция_член_функции> ::=
<име_на_тип> <име_на_клас>::<име>({<списък_формални_параметри>}опц) {
  <тяло>
}
```

КЛАСОВЕ

- Дефиниция на клас

```
class <име_на_клас> {
  {<спецификатор_за_достъп>:}опц
  <член_данни>;|{<член_данни>;}опц

  {<спецификатор_за_достъп>:}опц
  {<член_функции>;|{<член_функции>;}опц}опц
};
```

- Член-функции

Пример:

```
class Point2D {
  private:
    double x;
    double y;
  public:
    void print() const;
};
void Point2D::print() const {
  std::cout << x << " " << y;
}
```

КЛАСОВЕ

- Дефиниция на клас

```
class <име_на_клас> {
  {<спецификатор_за_достъп>:}опц
  <член_данни>;|{<член_данни>;}опц

  {<спецификатор_за_достъп>:}опц
  {<член_функции>;|{<член_функции>;}опц}опц
};
```

- Константни член-функции
  - Имат същото поведение като в структурите
  - Не позволяват промяна на член-данните на класа
  - Това се указва чрез записването на запазената дума const в декларацията и в края на заглавието в дефиницията им
  - Извикват само const функции

КЛАСОВЕ

- Обекти - екземпляри на класа/структурата.

```
<дефиниция_на_обект_на_клас> ::=
<име_на_клас> <обект>{, <обект2>};
<обект>, <обект2> ::= <идентификатор>
```

Пример:

```
class Point2D {
  private:
    double x;
    double y;
  public:
    void print() const;
};

Point2D p1;
Point2D p2, p3;
```

КЛАСОВЕ

- Обекти

Пример:

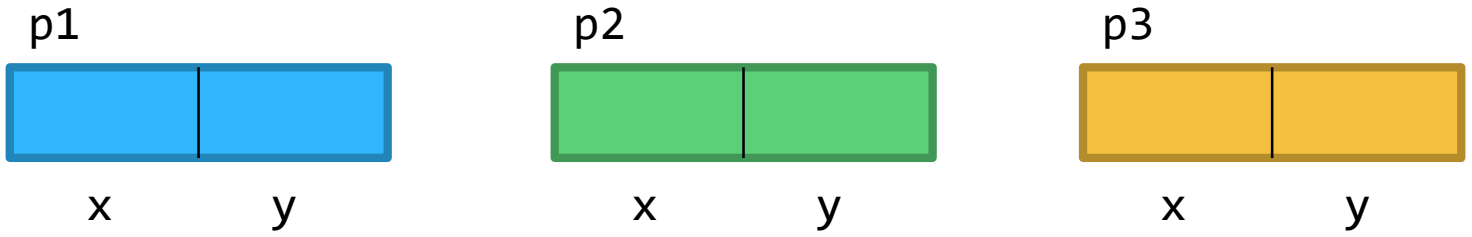
```
class Point2D {
  private:
    double x;
    double y;
  public:
    void print() const;
};

Point2D p1;
Point2D p2, p3;
```

Какво сме постигнали?

КЛАСОВЕ

- Памет (аналогично на структурите)
  - Памет за член-данните на класа се заделя при създаването на обект
  - Всеки обект разполага със собствени член-данни
  - Кодът на методите на класа НЕ се копира във всеки обект, а се намира само на едно място в паметта
  - Работят с указател this



КЛАСОВЕ

Достъп до компонентите на клас

- Вътрешен достъп - достъп на компонентите на класа до други компоненти на същия клас.

Член-функциите на клас имат пряк достъп до всички компоненти на класа без значение на секцията (спецификатор на достъп), в която се намират компонентите.

КЛАСОВЕ

Достъп до компонентите на клас

- Вътрешен достъп - достъп на компонентите на класа до други компоненти на същия клас.

Член-функциите на клас имат пряк достъп до всички компоненти на класа без значение на секцията, в която се намират компонентите.

Пример:

```
class Point2D {
  private:
    double x;
    double y;
  public:
    void print() const;
};
// Член-функция
void Point2D::print() const {
  std::cout << x << " " << y; // Има пряк достъп до x, y и т.н.
}
```

КЛАСОВЕ

Достъп до компонентите на клас

- Външен достъп - достъп до компоненти на клас чрез обекти на класа, дефинирани във външни за класа функции.

Външен достъп е възможен единствено до public компонентите на класа.

КЛАСОВЕ

Достъп до компонентите на клас

- Външен достъп - достъп до компоненти на клас чрез обекти на класа, дефинирани във външни за класа функции.

Външен достъп е възможен единствено до public компонентите на класа.

Пример:

```
class Point2D {
  private:
    double x;
    double y;
  public:
    void print() const;
};
// Член-функция
void Point2D::print() const {
  std::cout << x << " " << y;
}
// …
Point2D p1;
p1.print();         // Има пряк достъп до print (public секция)
std::cout << p1.x; // Няма пряк достъп до x (private секция)
```

КЛАСОВЕ

- Дефиниция на клас

```
class <име_на_клас> {
  {<спецификатор_за_достъп>:}опц
  <член_данни>;|{<член_данни>;}опц

  {<спецификатор_за_достъп>:}опц
  <член_функции>;|{<член_функции>;}опц
};
```

- Спецификатор за достъп
  - `public` – позволява външен достъп
  - `private` – забранява външен достъп
  - `protected` – забранява външен достъп

КЛАСОВЕ

- **Капсулиране на информация**

Процесът на скриване на информация се нарича капсулиране на информация.

Област на клас

- **Деклариран глобално** - започва от декларацията и продължава до края на програмата.

- **Деклариран локално** (вътре във функция или в тялото на клас)

** Не е възможно функциите да се дефинират в тялото на функцията или класа.*
Ще се получат функции, дефинирани във функция, което не е възможно.

Решение: Всички негови член-функции трябва да са вградени (inline).

Разделна компилация

- Класове и функции
  - Често реализацията на класовете (някои функции) се разделя в два файла – .cpp && .h
  - Хедърните файлове съдържат дефиницията на даден клас
  - Реализацията на класа попада в отделен cpp файл.
  - Този механизъм се нарича разделна компилация и предоставя възможност да се компилират само класовете, които са били променени.
  - Това не е възможно ако дефиницията и декларацията на класа се намират в един файл.

Допълнителна литература:
http://www.math.uaa.alaska.edu/~afkjm/csce211/handouts/SeparateCompilation.pdf

Разделна компилация

- Възможно ли е двойно включване на хедърни и файлови директиви – guards

РАЗДЕЛНА КОМПИЛАЦИЯ

- #**define** - предпроцесорна директива, която позволява да се дефинира макрос в изходния код.
  Позволява да се зададе име на константа преди програмата да се компилира.

  Пример:
  ```
  #define identifier replacement
  ```

Разделна компилация

- #**ifndef** - проверява дали вече съществува подобна дефиниция в същия или във включен файл.
  Ако дефиницията не съществува, кодът между #**ifndef** и #else или #endif се добавя.

```
// header.h
#ifndef HEADER_H_
#define HEADER_H_
// Code placed here is included only once
#endif
```

Разделна компилация

- #**pragma once** - нестандартизирана предпроцесорна директива, която се използва, за да се забрани многократното включване на един и същи файл.

Пример:

```
// header.h
#pragma once
// Code placed here is included only once
```

Екв.

```
// header.h
#ifndef HEADER_H_
#define HEADER_H_
// Code placed here is included only once
#endif
```

РАЗДЕЛНА КОМПИЛАЦИЯ

- **#pragma once** – нестандартизирана предпроцесорна директива, която се използва, за да се забрани многократното включване на един и същи файл.

Недостатъци

- Не може да работи със сложни пътища
  - Компилаторът може да не различи два пътя с едно и също име на файл
  - Възможно е да не различи, че два пътя се отнасят до един и същ файл
- Няма защитните механизми на #ifndef.
- Нестандартизиран – компилатори и поведение

ВРЕМЕ ЗА ВАШИТЕ ВЪПРОСИ

?