Теми за държавен изпит

17. Обектно-ориентирано програмиране. Подтипов и параметричен полиморфизъм. Множествено наследяване.

Шаблони:

Шаблоните в C++ позволяват дефинирането на "общи" функции и класове, които работят с неопределени типове по общ, унифициран начин.

Пример:
```
template <typename T>
class Point {
    T x,y;
    ---
    void translate (T a) {
        x += a;
        y += a;
    }
};
```

Типът T може да бъде заместен с произволен тип, който съдържа операцията +=.

- шаблони на функции

```
template <typename <параметър> {, typename <параметър>} >
    <сигнатура> { <тяло> }
```

Типовите параметри могат да имат стойности по подразбиране и могат да участват в тялото, връщания резултат и типовете на параметрите.

пример:
```
template <typename T>
void swap (T& a, T& b) { --- }
```

Шаблонът не се компилира. При всяко използване с различни типове се генерира нова функция, която се компилира.

- шаблони на класове

template <typename <параметър> {, typename <параметър>} >
class <име> { <тяло> };

Типовите параметри могат да се използват за полета, параметри на методи, за връщан резултат от методи и в тялото на методи.

Пример: Point <int> p;

Статично и динамично свързване:

- Статично – прави се сравнение м/у формални и фактически параметри и се избира най-точното съвпадение. Методът, който ще се извика се определя по време на компилация и при всяко изпълнение е един и същ. В C++ по подразбиране свързването е статично.

- Динамично – методът, който ще се извика, се определя по време на изпълнение.

Ако имаме следния пример:

Player* pp = new Hero ("Arthur", 20, 100);
pp->print();

При статично свързване ще се извика Player::print, а при динамично – методът на този клас, от който е обектът, към който сочи указателя (т.е. Hero::print). Можем да указваме какво е свързването за всяка отделна член-функция.

Виртуални функции:

Декларира
Функция, за която свързването е динамично, се нарича виртуална. ~~Дефинира~~
се така: virtual <сигнатура>;

Класове с виртуални функции се нарича полиморфни.

пример: class Player { ... virtual void print() const; ... };

Player* pp = ~~new Player(...);~~ new Player(...);
Player* ph = ~~new Hero(...);~~ new Hero(...);

pp -> print(); // Player::print();
ph -> print(); // Hero::print()

• Особености
- само методи могат да бъдат виртуални.
- конструкторите не могат да са виртуални
- наследяваната чл-ф-я в производния клас трябва да е със същата сигнатура
- видимостта на виртуален метод се определя от видимостта ѝ в ~~ф~~ класа на указателя, през който се извиква.

Извикване на виртуален метод:
• чрез обект - Player p; p.print()
Тогава свързването е статично, типът се знае предварително
• чрез указател
Player * pp = &h; pp->print();
свързването е динамично.

- чрез референция – Player& rp = h; rp.print();
Еквивалентно на указател, динамично свързване.

- чрез указване на област – Player::print();
свързването е статично, указваме сме точно кой метод се извиква.

- от метод – void Player::f(){ ... print(); ...}
Еквивалентно на извикване чрез this, динамично свързване

- от конструктор/деструктор на основен клас – статично

Не е задължително виртуална функция да има нова реализация във всеки производен клас.

Виртуален деструктор:

пример: Player* pp = new Hero; ... delete pp;
Понеже свързването е статично, то ще се извика деструкторът на Player.
Динамичната памет за Hero остава неосвободена и имаме тежък leak.
Можем да декларираме деструктора като виртуален и тогава ще се извиква правилният.

Полиморфизъм:

Един от четирите основни принципа на ООП. Той е свойството на даден елемент да приема повече от една форма.

Видове полиморфизъм:
- претоварване на функции – различна реализация в зависимост от типовете и броя на параметрите.
- параметричен полиморфизъм (шаблони) – една и съща реализация независимо от типовете

• Подтипов полиморфизъм (наследяване + virtual) – една и съща сигнатура в всички подтипове за даден тип (с евентуално различна реализация).

Интерфейс:

Множество от операции, които поддържа даден тип. Не включва реализация на операциите, а само техните ~~имена~~ сигнатури. Ако няколко класа имат общ интерфейс, с тях може да се работи унифицирано.

Абстрактни класове:

Наследяването на интерфейси в C++ се реализира чрез чисти виртуални функции. Декларират се така: virtual <сигнатура> = 0;
Така се освобождаваме от задължението да представим дефиниция за метода.
Клас, който има поне една чиста виртуална функция се нарича абстрактен.

Особености на абстрактните класове:

- Не можем да създаваме обекти от тях.
- Ако (~~на следващ~~) наследник на (~~абстрактен~~) абстрактен клас не реализира някоя чиста вирт. ф-я, той също става абстрактен.
- Могат да имат конструктори и деструктори, но се викат косвено от производен клас.

Масиви от обекти и хетерогенни контейнери:

Можем да дефинираме масиви от обекти от един и същи клас.

<клас> <име>[<брой>]

пример: Player players[10];

players[0].print(); // достъп до елемент

Хетерогенните контейнери са контейнери, които съдържат обекти от различен тип. Реализират се чрез указател към полиморфен тип. Над хетерогенните контейнери могат да се изпълняват масови операции от общия интерфейс.

пример:

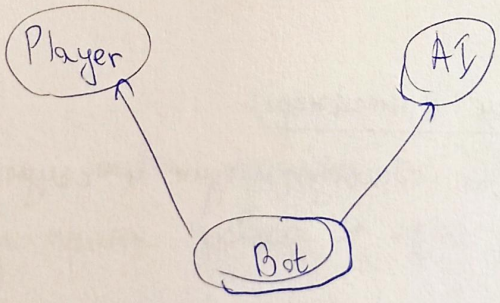
```
Player* players[3];
players[0] = new Player (...);
players[1] = new Hero (...);
players[2] = new Bot (...);
for (int i=0; i<3; i++){
    players[i].print(); // динамично свързване
}
```

Множествено наследяване:

В C++ един производен клас може да има повече от един основен клас.

пример:

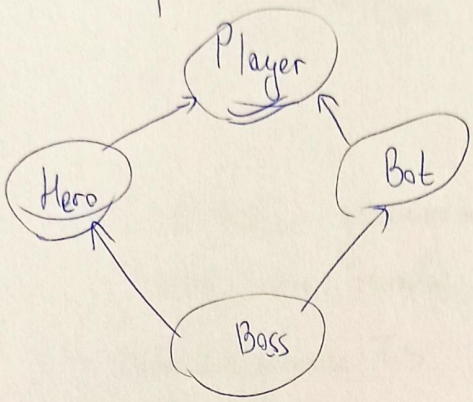
```
class Bot : public Player, public AI {...};
```



Конструкторът на производния клас трябва да указват как се конструират всяка една от наследените части. Деструкторите на основните класове се викат автоматично.

Проблеми при множественото наследяване:

- усложнява се йерархията
- диамантен проблем



Всяка компонента на Player се повтаря в Boss, защото се взема веднъж от Hero и веднъж от Bot. Искаме да разрешим нееднозначността като поддържаме единствено копие на общия прa-основен клас Player. Проблемът се решава с виртуални основни класове. (~~декларираме Player като виртуален~~) Това е във физическото представяне за Boss той е споделен (а не се повтаря два пъти).

```
class Hero : virtual public Player {...}
class Bot : virtual public Player {...}
```

Един клас не може да е виртуален сам по себе си. Той може да бъде наследен ~~като виртуален~~ виртуално.

Още не е решен проблемът с повторението на операции. Boss::print() ще отпечатва една и същата Player част два пъти.

~~Пример за избягване на повторението:~~

```
void <клас>::printDirect(...) const {
<нещо>
```