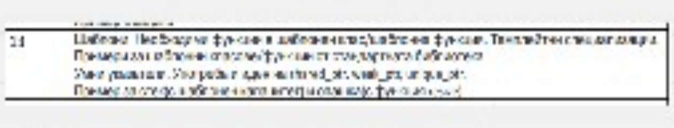
Тема 13



Шаблони

- параметричен полиморфизъм
- compile-time полиморфизъм

def. клас|ф-я с общо предназначение спрямо типа

генерира се нов код, където T е заменено със съответния тип

Синтаксис:

```
template <typename T>
const T& max(const T& lhs, const T& rhs){
        return lhs<rhs?rhs:lhs; }
```

→ можем да ограничим с какви типове може да се ползва шаблонът, например чрез class вместо typename

→ можем да добавиме идентификатори

def необходима функция - ф-ия, която всеки тип трябва да има, за да може да се използва темплейта

- Проблем с разделната компилация: .h не може да сигнализира на .cpp с какви типове да се генерира кода
  - I. .hpp - без разд. комп.
  - II. инстанциране на типове - обозначаваме в края на .cpp файла с какви типове да се генерира кода

Темплейтна специализация

def: клас/ф-я с различно поведение спрямо типа

```
template <class T>
  sort (T* arr, size_t size)
     quick sort + insertionsort

  template <>
  sort (char* arr, size_t size)
     count sort
```

sort <int>

sort <char>

Примери от стандартната библиотека:

vector, queue, stack
std::sort
умни указатели

Умни указатели

- обвиващ клас на указател, чийто деструктор се грижи за триенето на данните, към които сочи ук-ля

- Така не използваме new ⟷ delete
- Конструкторът приема ук-л към обект; дестр-ът извиква дестр. на обекта

1) auto_ptr - depricated (остарял и не се използва, но още работи, за да може да се използва стар код)

2) unique_ptr - точно един указател за точно един обект

- Синтаксис: unique_ptr<A> up = make_unique<A>(...)

  → вместо 'new'

- забранени К.К. и оп=, защото искаме да е един ⇒ трябва да има move семантики

! - Ще се счупи, ако към предварително създаден обект насочим два unique_ptr. Нито един от тях не подозира за съществуването на другия ⇒ при изтриването на единия паметта ще се изтрие и деструкторът на втория ще гръмне.

- Функцията make_unique<T>(parameters list) - шаблонна функция, която приема параметрите, нужни за създаването на T, извиква new и връща unique_ptr към обекта

3) shared_ptr - 1 обект, но много указатели

def: Клас, който пази ук-л към обект и брояч колко указателя са насочени към обекта.

При първият ук-л се заделя обектът.
При изтриването на последния ук-л се изтрива и самия обект (когато брояча стане 0)

- Синтаксис: shared_ptr<A> sp = make_shared<A>(2,3);

  // shared_ptr<A>(new A(2,3));

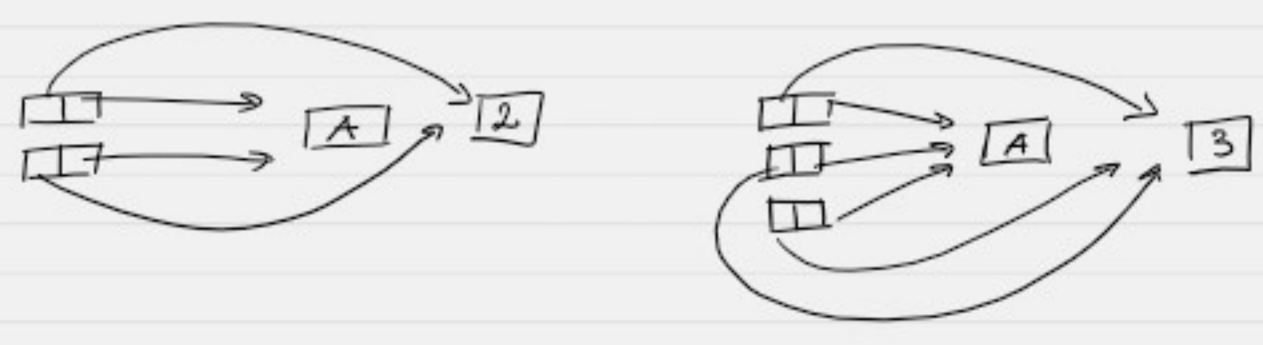
- shared_ptr има: предефинирани оператори * и →, копиране и местене

Триене:
- → трием ук-л → намаляваме брояча
- → последният ук-л изтрива обекта
- → при изтриването на последния ук-л, трябва да се изтрие и counter-а

Брояча не може да е static член данна, защото в случай, че имаме повече от един еднотипни обекти с насочени към тях shared_ptr броячът ще брои общо всички създадени shared_ptr

//shared_ptr::counter ==5

⇒ пазим ук-л към външна памет, която пази бройката.



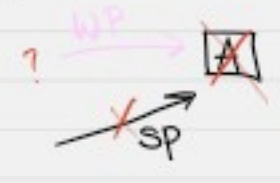
```
~shared_ptr() {
   refCount --;
if(*refCount == 0){
        delete data;
        delete refCount;
   }
}
```

- ф-ята make_shared<T>(..) - аналогично на make_unique<T>

  return shared_ptr<A>(new A(...));

4) weak_ptr - non-owning refference

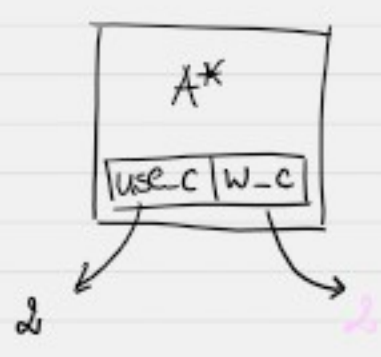
- ук-л към обект, който е менажиран от shared_ptr ⇒ не влияе на триенето и може да сочи към вече изтрит обект

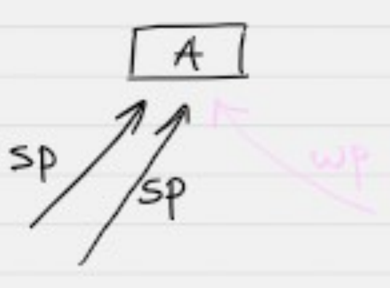


- възможно е промотиране в shared_ptr

weak_ptr трябва да има проверка дали обектът е изтрит

⇒ weak_ptr има брояч за броя на shared_ptr сочещи към обекта + брояч за броя на weak_ptr сочещи към обекта





use_count → брой на shared_ptr
weak_count - брой на weak_ptr + $\begin{cases} 1, & \text{use\_count} \geq 1 \\ 0, & \text{use\_count} = 0 \end{cases}$

Кога се трие обекта: use_count = 0
Кога се трие брояча: weak_count = 0