Тема 13 Шаблони. Необходими функции в шаблонен клас/шаблонна ф-ия. Темплейтни спецификации. Пример за шаблонни класове от стандартната библиотека. Умни указатели. Употреба и идея на Shared_ptr, Weak_ptr и unique_ptr.

Шаблон
- параметричен полиморфизъм (статичен)
- клас/ф-ия с общо предназначение спрямо типа

Пример:

```
template <typename T>
class Vector {...};
```

- ключова дума template

Необходими ф-ции в шаблонен клас/ф-ция
Имаме следната постановка

Пример:
```
template <typename T>
const T& max(const T& l, const T& r){
    return (l<r)?r:l;
}
```
Тук всички класове, за които се извиква max() трябва да имат предефиниран оператор <, иначе ще имат CompileTime error

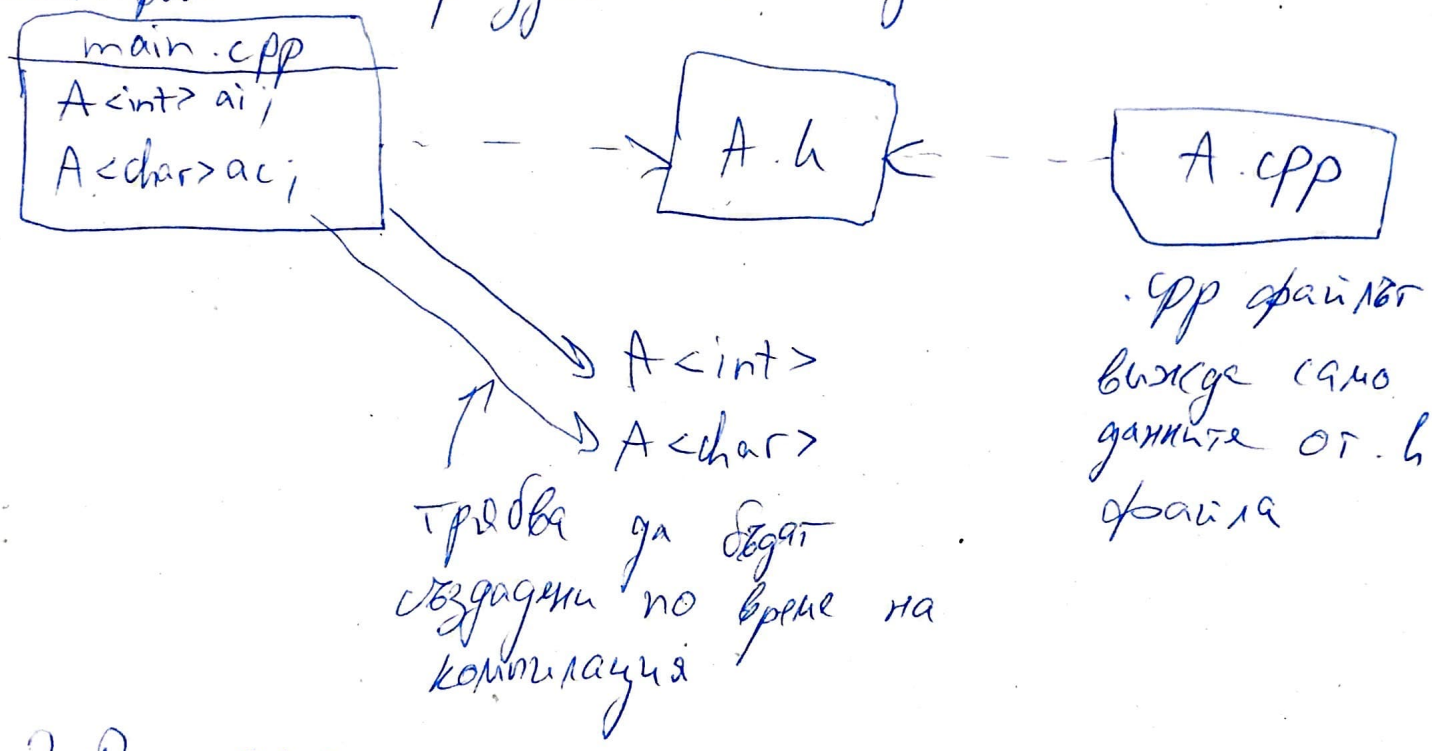
⚠ След C++... има методология за оказване на
защитените оператори

Компилатор замества typename-а с конкретна променлива.

Пример: max<int>(1,2);

↳ генерира се нов код за max и се замества T

⚠ Проблем с разделна компилация



2 Решения:

I) Разделяне на .h и .cpp файл

Като на последния ред на .cpp се казва, кои са типовете.

\+
разделната комп. (не трябва # да се компилира много пъти)

\-
ограничен брой предефинирани типове

Темплейтен клас .hpp

\+
не ограничен брой
различни типове

\- няма разделяне
компилация

Темплейтна спецификация
- клас / ф-ия с конкретно предназначение спрямо типа

Пример: std::vector<bool> Vect;
Не използва масиви а Bitset, за да спести памет.

Пример:
```
template <class T>
sort(T* arr, size_t)
    => using QuickSort

template <>  <- Темплейтна спецификация
sort(char* arr, size_t)
    => using (Counting Sort)
```

Умни указатели
Обзор дали по стандарт за вдигане на абстракция и избягването на ползването на new и delete

I auto_ptr

// legacy

Използва се чрез constructor за init и при изтриване се деалокира заделената памет.

Пример:

```
auto_ptr<A> ptr(new A());
```

II Unique_ptr

Парче памет е обвито от точно един определен pointer.

Използва се make_unique<..>(..) за да се избегне new-то.

Забранени са CC и OП=

Пример:

```
class A
    A(int, bool)
    ~A
```

```
int main()
{
    unique_ptr<A> up=
    make_unique<A>(2,true);
}
```

A() → ↑ ~A()

⚠ Edge case

```
{
    A* a = new A();
    Unique_ptr<A> up1 = make_unique<A>(a);
    Unique_ptr<A> up2 = make_unique<A>(a);
}
```

← ще гръмне, защото се опитва да изтрие празна данна

III. shared_ptr

Цел: имаме 1 спец. пакет и искаме много указатели към него.

Те трябва да бъдат заделени от 1 указател и чрез shallow копиране да се създават други.

Последния трябва да я изтрие.

Пример: shared_ptr<A> sp = make_shared<A>(2, true);

S_p<A> sp2 = sp;

S_p<A> sp3 = sp;

sp2.release();

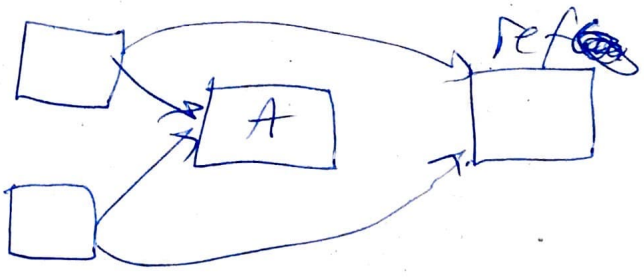
sp.release();

sp->f(); // примерно

Реализация

int* ref;

Последния изтрива тази памет.



IV. Weak_ptr

Използва се заедно с shared_ptr.

Този указател не отговаря за алокирането и деалокирането на данни а само наблюдава конкретен адрес в пакета.

Възможно е стойността на weak_ptr да е невалидна.

Реализация:

Counter* _counter

масив, съдържащ 2 int числа някъде в паметта.

При shared_ptr отговаря за изтриване на A и counter.

Weak_ptr отговаря за изтриване на counter.

Може да има 3 weak_ptr към несъществуващ обект и те няма как да знаят.

⚠ edge проблеми

- lock() - от weakPtr се създава SharedPtr, ако _data е валиден обект

Stack.hpp

```
#pragma once
#include <iostream>

template <class T, unsigned N>
class Stack
{
    T arr[N];
    unsigned size=0;
public:
    void push(const T&);
    void push(T&&);

    void pop();
    const T& peek() const;
    bool isEmpty() const;

};
template <class T, unsigned N>
void Stack<T,N>::push(const T& el)
{ if(size >= N)
      throw std::logic_error("Full Stack!");

    arr[size++] = el;
}
```

```
template <class T, unsigned N>
void Stack<T,N>::push(T&& el)
{ if(size >= N)
      throw std::logic_error("Full stack!");
  arr[size++] = std::move(el);
}
template <class T, unsigned N>
void Stack<T,N>::pop()
{ if(size <= 0)
      throw std::logic_error("Empty stack!");
  size--;
}
template <class T, unsigned N>
const T& Stack<T,N>::peek() const
{ return arr[size-1];
}
template <class T, unsigned N>
bool Stack<T,N>::isEmpty() const
{ return size==0;
}
```

Queue.hpp

```
#pragma once
#include <iostream>

template <class T>
class MyQueue
{ private:
    static const size_t DEFAULT_CAP=8;
    T* date =nullptr;
    size_t size=0;
    size_t cap= DEFAULT_CAP;

    size_t get=0;
    size_t put=0;

    void resize();

    void copyFrom(const MyQueue<T>& other);
    void moveFrom(MyQueue<T>&& other);
    void free();

    void incrementIndex(size_t& index);
public:
    MyQueue();
    MyQueue(const MyQueue<T>& other);
    MyQueue(MyQueue<T>&& other) noexcept;
    MyQueue<T>& operator=(const MyQueue<T>& other);
    MyQueue<T>& operator=(MyQueue<T>&& other) noexcept;
    ~MyQueue();
```

```
    void push(const T& obj);
    void push(T&& obj);
    void pop();
    const T& peek() const;
    bool isEmpty() const;

};
template <class T>
MyQueue<T>::MyQueue()
{ data = new T[cap];
}
template <class T>
void MyQueue<T>::push(const T& obj)
{
    if(size == cap)
        resize();
    data[put] = obj;
    incrementIndex(put);
    size++;
}
template <class T>
void MyQueue<T>::push(T&& obj) {
    if(size == cap)
        resize();
    data[put] = std::move(obj);
    incrementIndex(put);
    size++;
}
```

Queue.hpp

```
template <class T>
void MyQueue<T>::pop()
{
    if(isEmpty())
        throw std::logic_error("EmptyQueue");
    incrementIndex(get);
    size--;
}
template <class T>
void MyQueue<T>::resize()
{
    T* resizedData = new T[cap*2];
    for(size_t i=0; i<size; i++)
    {
        resizedData[i] = std::move(data[get]);
        incrementIndex(get);
    }
    cap*=2;
    delete[] data;
    data = resizedData;
    get=0
    put=size;
}
template <class T>
void MyQueue<T>::copyFrom(const MyQueue<T>& other)
{
    data = new T[other.cap];
    for(size_t i=0; i<other.cap; i++)
        data[i] = other.data[i];
    get=other.get;
    put=other.put;
    size=other.size;
    cap=other.cap;
}
```

```
template <class T>
void MyQueue<T>::moveFrom(MyQueue<T>&& other)
{
    get=other.get;
    put=other.put;
    size=other.size
    cap=other.cap;
    data=other.data;
    other.data=nullptr;
    other.size=other.cap=other.get=other.put=0;
}
template <class T>
void MyQueue<T>::free()
{
    delete[] data;
    data=nullptr;
    cap=size=get=put=0;
}
template <class T>
MyQueue<T>::MyQueue(const MyQueue<T>& other){
    copyFrom(other);
}
template <class T>
MyQueue<T>::MyQueue(MyQueue<T>&& other) noexcept{
    moveFrom(std::move(other));
}
template <class T>
MyQueue<T>& MyQueue<T>::operator=(const MyQueue<T>& other){
    if(this!=&other)
    {free();
     copyFrom(other);
    }return *this;
}
```

Queue.hpp

```
template <class T>
MyQueue<T>& MyQueue<T>::operator=(MyQueue<T>&& other) noexcept
{
    if (this != &other)
    {
        free();
        moveFrom(std::move(other));
    }
    return *this;
}
template <class T>
MyQueue<T>::~MyQueue()
{
    free();
}
template <class T>
void MyQueue<T>::incrementIndex(size_t& index)
{
    (++index) %= cap;
}
template <class T>
bool MyQueue<T>::isEmpty() const
{
    return size == 0;
}
template <class T>
const T& MyQueue<T>::peek() const
{
    if (isEmpty())
        throw std::logic_error("Empty queue!");
    return data[get];
}
```