Семинар #8: Умные указатели и классы представления

Проблема ошибок при работе с динамической памятью

Одни из частых ошибок, которые возникают при программировании на языке C и C++, являются утечки памяти. Утечка памяти возникает тогда, когда была выделена память в куче с помощью функции `malloc`, но эта память не была освобождена с помощью функции `free`. Аналогичная проблема возникает и при использовании операторов `new` и `delete`:

```
void func(int n)
{
    int* p = new int[n];
    // забыли сделать delete[] p;
}

int main()
{
    func(100); // Утечка
    func(200); // Ещё одна утечка
}
```

Например, в этой программе утечка памяти происходит в функции `func`, так как в ней выделяется память с помощью `new[]`, но не освобождается с помощью `delete[]`. После выхода из функции `func` мы не сможем освободить эту память, даже если бы хотели, так как мы не знаем чему равен указатель `p`.

Может показаться, что такую ошибку легко увидеть, но на самом деле, это не всегда верно. Программа может быть очень большой, выделений памяти может быть очень много, выделение памяти может происходить в одной функции, а освобождение этой памяти происходить в другой функции и т. д. Сложности добавляет использование исключений. Ведь если где-то в функции будет брошено исключение, то произойдёт выход из функции без освобождение выделенной в этой функции памяти. Рассмотрим, например, такую функцию, содержащую утечку памяти:

```
void func(int n)
{
    int* p = new int[n];
    int* q = new int[n];
    // ...
    delete[] q;
    delete[] p;
}
```

Утечка в данной функции произойдёт, если второй оператор `new` не сможет выделить память и бросит исключение. Произойдёт выход из этой функции без освобождения памяти на которую указывает указатель `p`.

Другая проблема, которая может возникнить при ручном выделении памяти, это проблема двойного удаления. Удаление объекта, который уже был удалён, приведёт к неопределённому поведению:

```
int* p = new int[10];
// ...
delete[] p;
// ...
delete[] p; // UB
```

Опять же, на таком простом примере эта ошибка очевидна, но в больших программах с большим количеством использований `new`/`delete` памяти такую ошибку может быть трудно заметить.

Хотелось бы иметь способ писать код так, чтобы минимизировать возможные ошибки.

Пишем свой умный указатель

Давайте используем знания, полученные при изучении C++ и напишем класс, который будет освобождать память за нас. Назовём его `SmartPointer`, то есть умный указатель. Умный, потому что он сам будет освобождать память. Этот класс должен конструироваться от обычного указателя, который указывает память в куче и освобождать память в деструкторе.

```
#include <iostream>

template <typename T>
class SmartPointer
{
private:
    T* pointer;

public:
    SmartPointer(T* pointer) : pointer(pointer) {}
    T& operator*()  const {return *pointer;}
    T* operator->() const {return pointer;}

    ~SmartPointer()
    {
        std::cout << "Deleting" << std::endl;
        delete pointer;
    }
};

int main()
{
    SmartPointer<int> p = new int(123);
    std::cout << *p << std::endl;
}
```

В этой программе нет утечки памяти, потому что `delete` вызовется в деструкторе объекта `q`.

Однако, у этой реализации умного указателя есть много проблем. Самая главная проблема заключается в том, что не были прописаны конструктор копирования и оператор присваивания. В результате конструктор копирования и оператор присваивания были созданы автоматически. Такие автоматически созданные методы будут просто копировать внутренний указатель в другой объект умного указателя. Это может привести к сеьёзным ошибкам:

```
int main()
{
    SmartPointer<int> p = new int(123);
    SmartPointer<int> q = p;
    // UB: двойной delete
}
```

Умный указатель `q` скопирует указатель в себя поле `pointer` указателя `p`. В результате оба умных указателя будут указывать на одно и то же место. Следовательно, в деструкторах обоих умных указателей будет вызван `delete` для одного и того адреса. Получилось двойное освобождение, что является неопределённым поведением.

Вообще, для данной реализации умного указателя мы не можем допустить того, чтобы несколько таких умных указателей указывали на один объект в куче. Ведь, если хотя бы один из этих умных указателей будет уничтожен (например, если мы выйдем из области видимости, где данный умный указатель был создан), то он уничтожит и объект в памяти, который в этот момент может использоваться через другие умные указатели. Это неопределённое поведение. Если же все такие умные указатели уничтожатся разом, то все они будут удалять один и тот же объект в своих деструкторах. Это тоже неопределённое поведение.

Есть два общепринятых способа избежать такой ситуации:

1. Сделать так, чтобы на один объект в куче мог указывать максимум только один умный указатель. Этот способ использует умный указатель `std::unique_ptr`.
2. Завести счётчик и подсчитывать сколько умных указателей указывает на данный объект в куче. Если один из указателей, указывающий на данный объект, уничтожается то счётчик уменьшается на 1. Объект уничтожается только тогда, когда счётчик достигнет нуля (то есть тогда когда уничтожится последний умный указатель, который указывал на данный объект). Этот способ использует умный указатель `std::shared_ptr`.

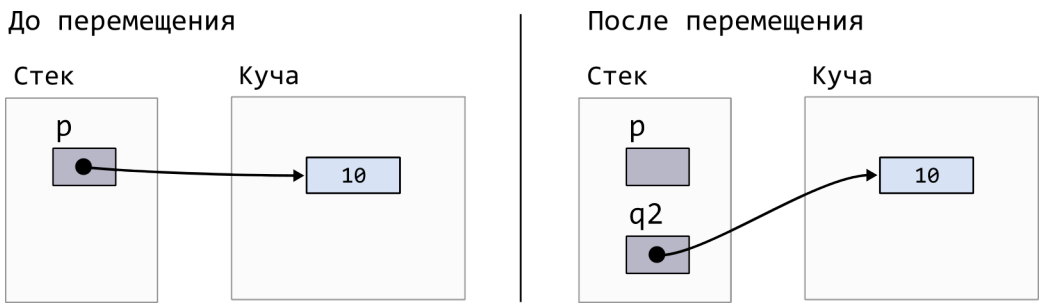
Умный указатель `std::unique_ptr`

`std::unique_ptr` – это умный указатель из библиотеки `memory`, который реализует эксклюзивное владение объектом. Этот умный указатель обладает следующими свойствами:

- `unique_ptr` может конструироваться из обычного указателя на объект в куче. При этом этот конструктор является `explicit`-конструктором, поэтому конструироваться из обычного указателя можно только с помощью прямой инициализации. Объект в куче уничтожается в деструкторе `unique_ptr`.
- Несколько указателей `unique_ptr` не могут указывать на один объект в куче. Это достигается благодаря тому, что эти указатели нельзя копировать, но можно перемещать. При перемещении, `unique_ptr`, из которого происходит перемещение, перестаёт указывать на объект и становится нулевым.
- Указатель `unique_ptr` может быть нулевым, то есть не указывать никуда. Нулевой `unique_ptr` можно получить, если сконструировать объект с помощью конструктора по умолчанию.

```cpp
#include <iostream>
#include <memory>

int main()
{
    std::unique_ptr<int> p(new int(10));

    std::unique_ptr<int> q1 = p;              // Ошибка, unique_ptr нельзя копировать
    std::unique_ptr<int> q2 = std::move(p);   // OK, но теперь p стал нулевым,
                                              // только q2 указывает на объект в куче
}
```

Процесс перемещения умного указателя `unique_ptr` представлен на следующем рисунке:



Методы класса `std::unique_ptr`

- `operator*` и `operator->` – работают аналогично таким же операторам обычного указателя.
- `operator bool()` – оператор приведения к типу `bool`:

```
std::unique_ptr<int> p(new int(10));
if (p)
    std::cout << "Yes" << std::endl;

std::unique_ptr<int> q;
if (q)
    std::cout << "No" << std::endl;
```

- `get` – возвращает обычный указатель на объект:

```
std::unique_ptr<int> p(new int(10));
int* q = p.get();
```

- `release` – после вызова этого метода `unique_ptr` перестаёт указывать на объект и становится нулевым. Объект, при этом, не удаляется. Возвращает указатель на объект.

```
std::unique_ptr<int> p(new int(10));
int* q = p.release();
delete q;
```

- `swap` – меняет местами значение двух `unique_ptr`.

```
std::unique_ptr<int> p(new int(10));
std::unique_ptr<int> q(new int(20));

std::cout << *p << " " << *q << std::endl;
p.swap(q);
std::cout << *p << " " << *q << std::endl;
```

Передача `std::unique_ptr` в функции

Обычно функции принимают `unique_ptr` по значению. Умный указатель в такие функции передаётся с использованием семантики перемещения.

```
#include <iostream>
#include <memory>

void func(std::unique_ptr<int> p)
{
    std::cout << *p << std::endl;
}

int main()
{
    std::unique_ptr<int> p(new int(10));
    func(std::move(p));

    if (p == nullptr)
        std::cout << "p in main is null" << std::endl;
}
```

Но можно передавать `unique_ptr` и по ссылке. Это менее предпочтительный способ, так как он может быть медленнее. Из-за того, что в функции `func`, чтобы доступиться до объекта в куче, придётся делать два шага: сначала нужно перейти по ссылке к указателю `unique_ptr`, а потом по указателю к объекту в куче.

```
#include <iostream>
#include <memory>

void func(std::unique_ptr<int>& p)
{
    std::cout << *p << std::endl;
}

int main()
{
    std::unique_ptr<int> p(new int(10));
    func(p);

    if (p == nullptr)
        std::cout << "p in main is null" << std::endl;
    else
        std::cout << "p = " << *p << std::endl;
}
```

Примеры ошибочного использования `unique_ptr`

Нужно понимать, что если `unique_ptr` конструируется от обычного указателя, то это должен быть указатель, который пришел из оператора `new` и никакой другой. В частности нельзя делать следующее:

- Конструироваться от адреса локальной переменной:

  ```
  int a = 10;
  std::unique_ptr<int> p(&a);  // Ошибка
  ```

  В этом случае, в деструктроре `unique_ptr` будет вызыван `delete` с адресом локальной переменной. Это конечно приведёт к неопределённому поведению.

- Конструироваться от адреса полученного из оператора `new[]`:

  ```
  std::unique_ptr<int> p(new int[100]);  // Ошибка
  ```

  В этом случае, в деструктроре `unique_ptr` будет вызыван `delete`, а не нужный нам оператор `delete[]`. Это также приведёт к UB.

Еще одна ошибка может возникнуть, если создать два умных указателя от одного обычного указателя.

```
int* raw = new int(10);
std::unique_ptr<int> p(raw);
std::unique_ptr<int> q(raw);
```

В этом случае один умный указатель создаётся независимо от другого и не может проверить, есть ли ещё один указатель, указывающий на тот же объект. Поэтому оба указателя `p` и `q` будут указывать на один и тот же объект в куче. Естественно, когда эти указатели уничтожатся, они будут уничтожать объект, на который они указывают и произойдёт двойное удаление одного объекта. Это UB.

Функция `std::make_unique`

Чтобы исключить возможность возникновения ошибочных ситуаций, перечисленных выше, для создания умных указателей `unique_ptr` следует использовать специальную функцию `make_unique`. Эта функция сама, внутри себя, создаёт объект в куче с помощью оператора `new`, создаёт указатель `unique_ptr` на этот объект и возвращает его. Использовать её нужно так:

```
std::unique_ptr<int> p = std::make_unique<int>(10);
```

Или ещё лучше так:

```
auto p = std::make_unique<int>(10);
```

Интересной особеностью функции `std::make_unique` является то, что она принимает на вход не уже сконструированный объект, а *аргументы его конструктора*. Это можно увидеть на следующем примере:

```
#include <iostream>
#include <string>
#include <memory>

class Book
{
private:
    std::string title {};
    int pages         {};
    float price       {};

public:
    Book(const std::string& title, int pages, float price)
        : title(title), pages(pages), price(price)
    {
        std::cout << "Constructor" << std::endl;
    }

    Book(const Book& b)
        : title(b.title), pages(b.pages), price(b.price)
    {
        std::cout << "Copy Constructor" << std::endl;
    }
};

int main()
{
    // Правильное использование
    auto p = std::make_unique<Book>("Harry Potter", 100, 200);

    // Неравильное использование
    // Будет вызван на один конструктор больше
    auto q = std::make_unique<Book>(Book("Harry Potter", 100, 200));
}
```

Это происходит потому что функция `make_unique` принимает аргументы по ссылке и конструирует объект внутри себя. Во втором же случае объект `Book` был создан до входа в функцию `make_unique` и в функции был создан заново через конструктор копирования.

Также заметьте, что `make_unique` принимает произвольное количество аргументов произвольных (возможно различных) типов и произвольных(возможно различных) категорий.

Умный указатель `std::shared_ptr`

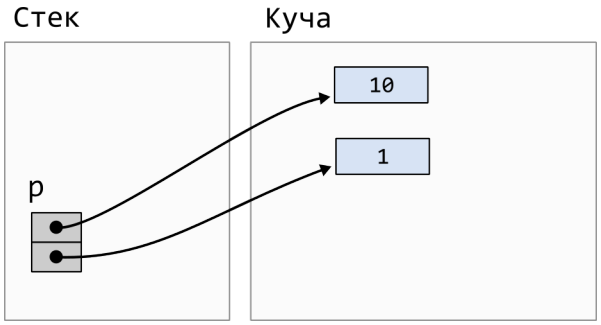
`std::shared_ptr` – это умный указатель из библиотеки `memory`, который реализует паттерн общего владения объектом. Этот умный указатель обладает следующими свойствами:

- `shared_ptr` может конструироваться из обычного указателя на объект в куче. При этом этот конструктор является `explicit`-конструктором, поэтому конструироваться из обычного указателя можно только с помощью прямой инициализации. `shared_ptr` сам корректно уничтожит объект в куче.
- Несколько указателей `shared_ptr` могут указывать на один объект в куче. Объект уничтожится только тогда, когда уничтожится последний `shared_ptr`, указывающий на этот объект.
- Указатель `shared_ptr` можно копировать и присваивать. При копировании создастся новый указатель `shared_ptr`, который будет указывать на тот же объект.
- Указатель `shared_ptr` можно перемещать. При перемещении, `shared_ptr`, из которого происходит перемещение, станет пустым.
- Указатель `shared_ptr` может быть нулевым, то есть никуда не указывать. Нулевой `shared_ptr` можно получить, если сконструировать объект с помощью конструктора по умолчанию.

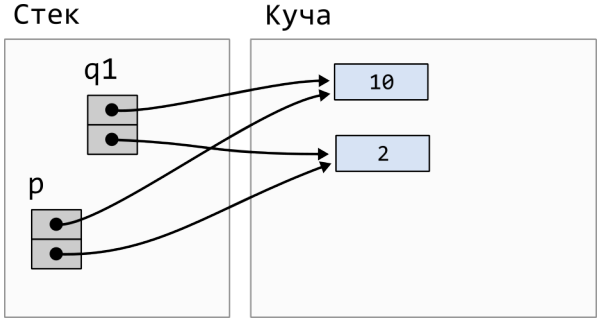
```cpp
#include <iostream>
#include <memory>

int main()
{
    std::shared_ptr<int> p(new int(10));

    std::shared_ptr<int> q1 = p;              // OK, теперь 2 указателя указывают на объект
    std::shared_ptr<int> q2 = std::move(p);   // OK, но теперь p стал нулевым,
                                              // q1 и q2 указывает на один объект в куче
}
```

Разберём этот участок кода по шагам, при этом будем смотреть, что происходит в памяти. Внутри объекта `shared_ptr` хранится указатель на объект в куче, а также указатель на счётчик ссылок.

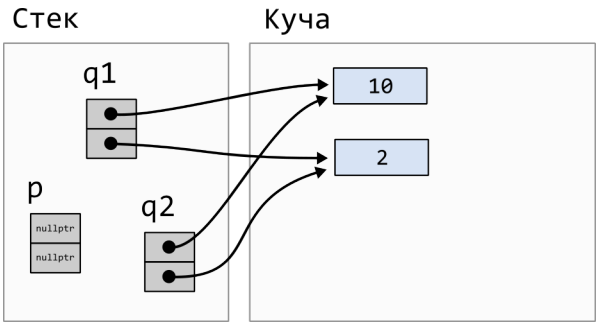
```cpp
// Создаём новый указателя shared_ptr
std::shared_ptr<int> p(new int(10));
```




```cpp
// Создаём копию указателя
std::shared_ptr<int> q1 = p;
```



```
// Перемещаем указатель p в q2
std::shared_ptr<int> q2 = std::move(p);
```



Методы класса shared_ptr

- `operator*` и `operator->` – работают аналогично таким же операторам обычного указателя.
- `operator bool()` – оператор приведения к типу `bool`:

```
std::shared_ptr<int> p(new int(10));
if (p)
    std::cout << "Yes" << std::endl;

std::shared_ptr<int> q;
if (q)
    std::cout << "No" << std::endl;
```

- `get` – возвращает обычный указатель на объект:

```
std::shared_ptr<int> p(new int(10));
int* q = p.get();
```

Передача класса `shared_ptr` в функции

Примеры ошибочного использования `shared_ptr`

Функция `std::make_shared`

Циклические ссылки

Указатель `std::weak_ptr`

Пользовательский удалитель (*custom deleter*) для умных указателей

Класс представления `std::string_view`

Класс представления `std::span`