

Жизненный цикл объекта. RAII

Амиран Мстоян

AI System Engineer (C++) в Huawei
Moscow Research Center



Проверка связи





Если у вас нет звука:

- убедитесь, что на вашем устройстве и на колонках включён звук
- обновите страницу вебинара (или закройте страницу и заново присоединитесь к вебинару)
- откройте вебинар в другом браузере
- перезагрузите компьютер (ноутбук) и заново попытайтесь зайти



Поставьте в чат:

-  если меня видно и слышно
-  если нет

Амиран Мстоян

О спикере:

AI System Engineer (C++) в Huawei Moscow Research Center

Математик-алгоритмист лаборатории научного центра при МФТИ



Вспоминаем прошрое занятие

Вопрос: что такое препроцессинг?



Вспоминаем прошрое занятие

Вопрос: что такое препроцессинг?

Ответ: это обработка исходников, при которой происходят макроподстановки, добавление хэдеров, замена комментариев



Вспоминаем прошное занятие

Вопрос: что такое компиляция?



Вспоминаем прошрое занятие

Вопрос: что такое компиляция?

Ответ: преобразование кода после
препроцессинга в машинный



Вспоминаем прошрое занятие

Вопрос: что такое линковка?



Вспоминаем прошрое занятие

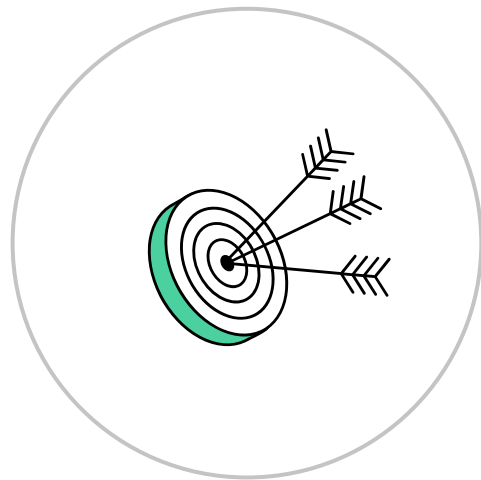
Вопрос: что такое линковка?

Ответ: объединение всех объектных файлов
в единый исполняемый файл



Цели занятия

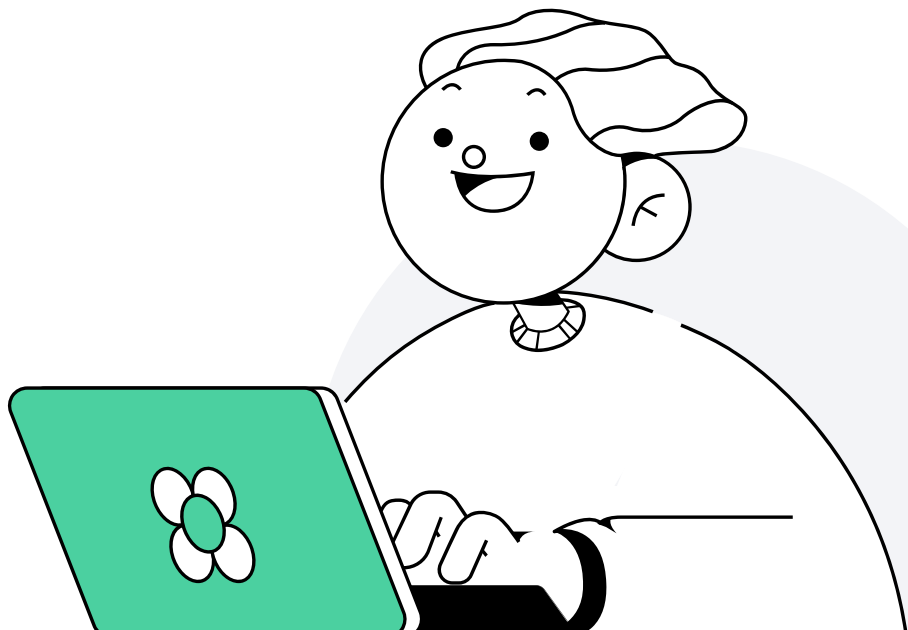
- Узнаем, что такое жизненный цикл объекта
- Узнаем, что такое идиома RAI
- Научимся использовать идиому RAI в своих программах



План занятия

- 1 Жизненный цикл объекта
- 2 Идиома RAII
- 3 Домашнее задание

*Нажми на нужный раздел для перехода



Жизненный цикл объекта



1

Жизненный цикл объекта

Любой объект в программе создаётся, “живет”, удаляется. Это и есть 3 стадии жизненного цикла объекта. Рассмотрим эти стадии более детально.

1

Создание

Выделение памяти под
существующий объект

2

Жизнь

Проводим какие-то действия
над объектом

3

Удаление

Высвобождение ресурсов,
выделенных под объект



Создание объекта - выделение памяти под наш объект

Память может выделяться двумя способами: на стеке и из кучи

Создание объекта

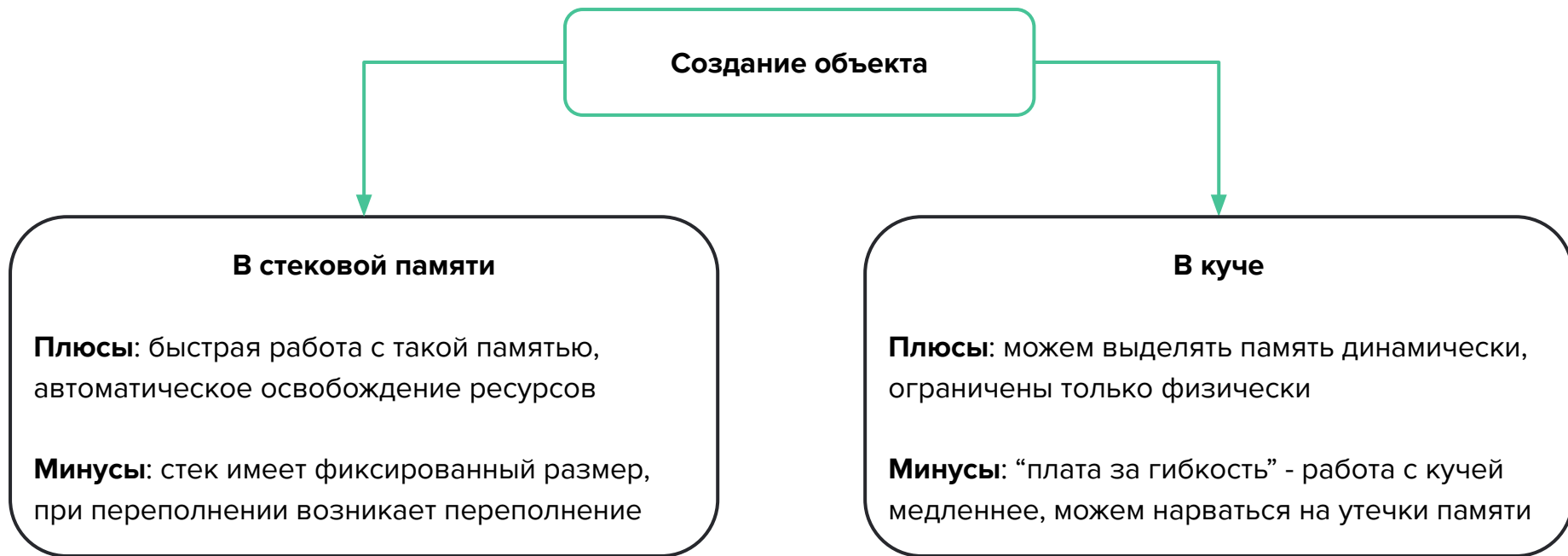
① В стековой памяти

- Оперативная память, организованная по принципу **LIFO** (last in, first out: последний добавленный в стек кусок памяти будет первым в очереди на вывод из стека)

② В куче

- Оперативная память, которая допускает динамическое выделение памяти: куча - как склад переменных. Взаимодействие с кучей осуществляется с помощью указателей

Создание объекта





Блок кода - это фрагмент программы, ограниченной фигурными скобками

Память на стеке

Если внутри блока кода объявляется переменная, то она помещается в стековую память.

Эта переменная будет “жить” до конца этого блока кода. Это значит, что как только мы выйдем из этого блока кода, вызовется деструктор для этой переменной, который и высвобождает занятую ей память. Такие переменные называются **автоматическими**.

Следует заметить, что порядок удаления таких переменных обратен порядку их создания (как вы понимаете из-за структуры стека).

Память на стеке. Пример

В данном примере сначала выделится память на строку, а потом на объект класса `my_class1`, затем на `my_class2`

```
void some_function(int a, int b)
{
    std::string s;
    // some code here
    my_class1 class1;
    my_class2 class2;
}
```

Память на стеке. Пример

И как бы ни вышли из этого блока (даже если возникнут исключения), то память автоматически очистится:

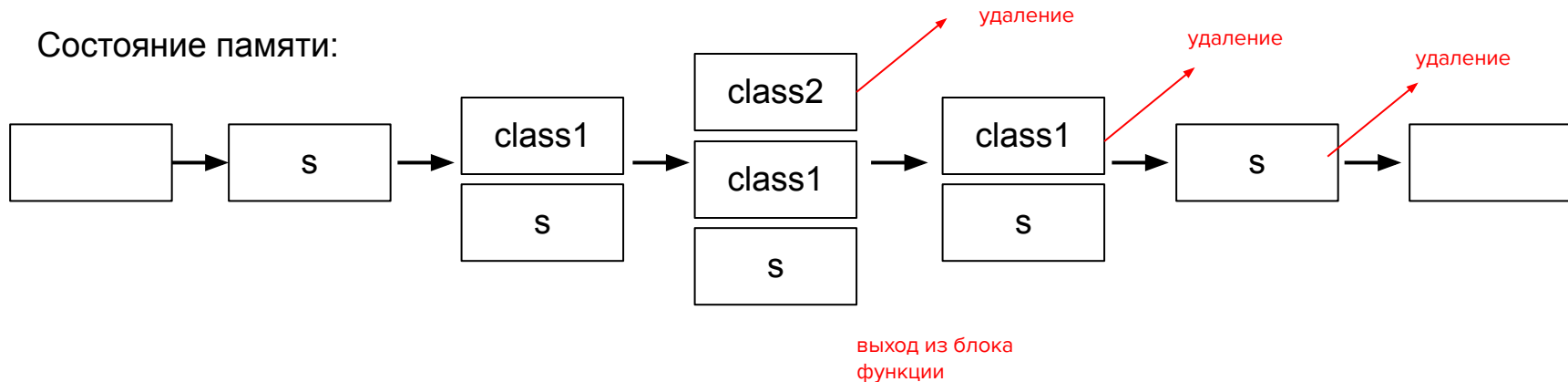
- сначала вызовется деструктор для объекта класса `my_class2`
- потом для объекта класса `my_class1`, а потом для объекта строки

```
void some_function(int a, int b)
{
    std::string s;
    // some code here
    my_class1 class1;
    my_class2 class2;
}
```

Почему такая память называется стеком?

```
void some_function(int a, int b)
{
    std::string s;
    // some code here
    my_class1 class1;
    my_class2 class2;
}
```

Состояние памяти:



Память из кучи

Время жизни переменных, созданных в динамической памяти управляется программистом.

Создаются они с помощью конструкций **new**, удаляются с помощью **delete**.

Вот в этом случае могут возникнуть проблемы, связанные с утечками памяти: необходимо внимательно следить, чтобы все ресурсы были корректно освобождены.

Память из кучи. Пример

В примере ниже, если мы сами не вызовем оператор **delete**, то память, выделенная на массив **arr** не будет автоматически освобождена

```
void some_func() {  
    int* arr = new int[10];  
    // some code here  
}
```

Память из кучи. Пример 2

Рассмотрим еще одну ситуацию, когда может произойти утечка памяти.

```
void some_func() {  
    int* arr = new int[10];  
    func(...);  
    // some code here  
    delete[] arr;  
}
```

Если в функции func возникнет исключение или в любом другом месте до вызова delete возникнет исключение, то блок кода покинется аварийно, и вызова функции delete не произойдет и возникнут утечки памяти.

Идея RAII призвана помочь программистам в таких ситуациях.

Идиома RAII



2



RAII - Resource Acquisition Is Initialization. Получение ресурса есть инициализация

Под ресурсом понимается то, что “берём напрокат” у операционной системы”: память, файл и т.д.



Основная идея: с каждым фактом захвата ресурса связывать какую-нибудь переменную на стеке (автоматическую переменную)

Такая переменная будет возвращать ресурс, когда он не нужен, с помощью своего деструктора

RAII

Рассмотрим следующий код в стиле C.

Вопрос: Какие проблемы вы видите?

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstdio>

int main() {

    FILE* f = fopen("out.txt", "w");

    if (f != nullptr) {
        fprintf(f, "hello world!\n");
        // some code here
        fclose(f);
    }
    else {
        printf("file open failed\n");
    }

    return 0;
}
```



RAII

Ответ: необходимость вручную закрывать файл и освобождать ресурсы

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstdio>

int main() {

    FILE* f = fopen("out.txt", "w");

    if (f != nullptr) {
        fprintf(f, "hello world!\n");
        // some code here
        fclose(f);
    }
    else {
        printf("file open failed\n");
    }

    return 0;
}
```



RAII

```
class file {
private:
    FILE* f;
public:
    file(const std::string& filename) {
        f = fopen(filename.c_str(), "w");
        if (f == nullptr) {
            throw std::runtime_error("file open failed");
        }
    }
    void write(const std::string& str) {
        fprintf(f, str.c_str());
    }
    ~file() {
        fclose(f);
    }
};

int main() {
    try {
        file f("out.txt");
        f.write("hello world\n");
    }
    catch (...) {
        printf("file open failed\n");
    }
    return 0;
}
```

RAII здесь была реализована с помощью простой “обертки”: класса, который управляет записью в файл

Как видим **теперь нет необходимости закрывать вручную файл**: при уничтожении класса file вызовется деструктор класса, который и выполнит необходимую работу за нас

Итоги



Итоги занятия

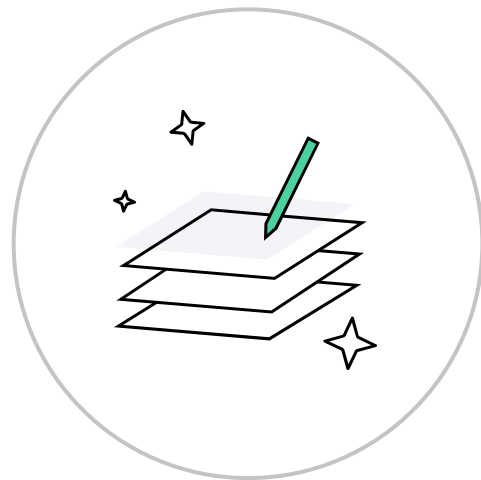
- 1 Узнали, что такое жизненный цикл объекта в C++
- 2 Познакомились с идиомой RAII



Домашнее задание

Давайте посмотрим ваше [домашнее задание](#).

- 1 Вопросы по домашней работе задавайте в чате группы
- 2 Задачи можно сдавать по частям
- 3 Зачёт по домашней работе ставят после того, как приняты все задачи



Дополнительные материалы

- [Статья](#) про RAI



Задавайте вопросы и пишите отзыв о лекции

Амиран Мстоян

AI System Engineer (C++) в Huawei
Moscow Research Center

