Конструкторы и деструкторы

Мотивация

Все ли вам нравится в интерфейсе стека написанного ранее?

```
class Stack {
  int* buffer;
  size_t size;
  static const size_t capacity = 100;

public:
  void Init();
  void Finalize();
  void Push(int value);
  void Pop();
  int& Top();

  int Top() const;
  size_t Size() const;
  bool Empty() const;

  static size_t Capacity();
};
```

Мотивация: инициализация

Все еще необходимо вручную инициализировать объект:

```
Stack stack;
stack.Init();  // без этой строки стек не заработает
```

А как проинициализировать константу?

```
const Stack stack;  // empty forever

// методы логически не константные!
// stack.Init()
// stack.Push()
```

Мотивация: копирование

При копировании создается побитовая копия!

```
Stack stack;
stack.Push(1);

auto other = stack;
other.Push(2);

stack.Size() == 2;  // wat
stack.Top() == 2;   // please, no
```

А еще непонятно, кто должен очищать память (вызывать `Finalize()`).

Мотивация: удаление

После работы со стеком необходимо самостоятельно очищать ресурсы:

```
Stack stack;
stack.Init();
// ...
stack.Finalize();  // утечка памяти, если не вызвать
```

Еще более страшная ситуация:

```
Stack<Stack<int>> stack;
// ...

// завершаем работу
while (!stack.Empty()) {
  stack.Top().Finalize();
  stack.Pop();
}
stack.Finalize();
```

Конструкторы

Конструктор

- **Конструктор** - особый метод класса, который вызывается при создании объекта.
- Этот метод не имеет возвращаемого значения.
- Его имя *совпадает* с именем класса.
- Конструктор вызывается *неявно* при создании объекта.

```
class Stack {
 public:
  Stack() {
    buffer_ = new int[kCapacity];
    size_ = 0;
  }
  // ... (other methods)
};

Stack stack;  // Вызывается конструктор Stack::Stack()
```

Конструктор

Существует несколько видов конструкторов:

- Параметрический конструктор
- Конструктор преобразования
- Конструктор по умолчанию
- Конструктор копирования
- Конструктор перемещения (будет рассмотрен в курсе позднее)

Виды конструкторов: параметрический конструктор

Параметрический конструктор

Параметрический конструктор - конструктор, который может быть вызван с несколькими аргументами (> 1).

```
class Stack {
 public:
  Stack(size_t size, int value) {  // стек из size элементов value
    buffer_ = new int[kCapacity];
    size_ = size;
    for (size_t i = 0; i < size_; ++i) {
      buffer_[i] = value;
    }
  }
  // ... (other methods)
};

Stack stack(10, 1);
// или Stack stack = Stack(10, 1);
// или auto stack = Stack(10, 1);
stack.Size();  // 10
stack.Top();   // 1
```

Списки инициализации

Списки инициализации

Проблема: давайте попробуем создать объект класса с константными полями и полями-ссылками с помощью параметрического конструктора.

```
class B {
  const int x_;
  double& y_;
public:
  B(int x, double& y) {
    x_ = x;
    y_ = y;
  }
};

double z = 0.0;
B b(0, z);
```

Списки инициализации

Проблема: давайте попробуем создать объект класса с константными полями и полями-ссылками с помощью параметрического конструктора.

```
class B {
  const int x_;
  double& y_;
public:
  B(int x, double& y) {
    x_ = x;
    y_ = y;
  }
};

double z = 0.0;
B b(0, z);  // Compilation error
```

```
error: uninitialized const member in 'const int'
error: uninitialized reference member in 'double&'
error: assignment of read-only member 'B::x_'
```

Списки инициализации

- Дело в том, что в момент выполнения тела конструктора все поля уже должны быть проинициализированы.
- Следовательно, надо инициализировать поля до входа в тело конструктора.

Списки инициализации

Требуемая синтаксическая конструкция называется списком инициализации.

```
class B {
  const int x_;
  double& y_;

public:
  B(int x, double& y) : x_(x), y_(y) {  // инициализируем
    // Работаем с инициализированными данными
  }
};

double z = 0.0;
B b(0, z);  // Ok
```

Если список инициализации пуст, то все поля инициализируются по умолчанию.

Если какое-то поле не проинициализировано, компилятор попытается проинициализировать его самостоятельно.

Списки инициализации

- *Важное правило*: порядок создания полей определяется порядком их объявления в классе, а не порядком в списке инициализации

```
struct A {
  int x;
  int y;
  int z;

  A(int value) : z(value), y(z), x(y) {
    // Предполагается, что сначала инициализируется z (=value),
    // затем y (=z), а затем x (=y)
  }
};

A a(11);
// Но на деле все обстоит иначе
std::cout << a.x << ' ' << a.y << ' ' << a.z << '\n';
```

```
1392458763 -452656 11
```

Списки инициализации

- *Важное правило*: порядок создания полей определяется порядком их объявления в классе, а не порядком в списке инициализации

```
struct A {
  int x;
  int y;
  int z;

  A(int value) : x(value), y(x), z(y) {
    // Предполагается, что сначала инициализируется x (=value),
    // затем y (=x), а затем z (=y)
  }
};

A a(11);
// Так и происходит
std::cout << a.x << ' ' << a.y << ' ' << a.z << '\n';
```

```
11 11 11
```

Виды конструкторов: конструктор преобразования

Конструктор преобразования

- *Конструктор преобразования* - конструктор, который может принимать ровно 1 аргумент.
- Данный конструктор используется для неявных (или явных) преобразований.
- Стандарт C++11 расширил понятие "конструктор преобразования", но это уже совсем другая история.

```
Stack::Stack(size_t size) : buffer_(new int[kCapacity]{}), size_(size) {
}

Stack stack(1);      // стек размера 1
Stack another = 3;   // стек размера 3

void f(Stack arg);

f(stack);  // Ok
f(10);     // Ok <=> f(Stack(10))
```

Ключевое слово `explicit`

```
Stack another = 3;  // стек размера 3
f(10);              // Ok <=> f(Stack(10))
```

Чтобы запретить подобные неявные (implicit) преобразования, необходимо попросить, чтобы они выполнялись только явно (explicit).

```
explicit Stack::Stack(size_t size) { ... }

Stack stack(1);     // Ok: явное преобразование
Stack another = 3;  // CE: неявное преобразование

void f(Stack arg);
f(stack);  // Ok: нет преобразования
f(10);     // CE: неявное преобразование
```

```
error: conversion from 'int' to non-scalar type 'Stack' requested
error: could not convert '10' from 'int' to 'Stack
```

Виды конструкторов: конструктор по умолчанию

Конструктор по умолчанию

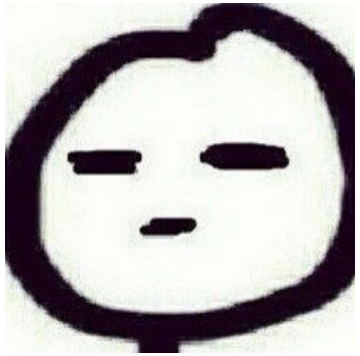
- *Конструктор по умолчанию* - конструктор, который может быть вызван без аргументов.

```
Stack::Stack() : buffer_(new int[kCapacity]), size_(0) {
}

Stack stack;
// или auto stack = Stack();

// Stack other();  CE: компилятор считает это объявлением функции
```

Вопрос от телезрителя

Если объекты без инициализаторов создаются с помощью конструктора по умолчанию, то как у нас создавался стек раньше?

```
class Stack {
public:
  void Init() { ... }
  // ...
};

Stack stack;  // <-- как?
stack.Init();
// ...
```



Вопрос от телезрителя

Если объекты без инициализаторов создаются с помощью конструктора по умолчанию, то как у нас создавался стек раньше?

```
class Stack {
public:
  void Init() { ... }
  // ...
};

Stack stack;  // <-- как?
stack.Init();
// ...
```

Ответ: с помощью конструктора по умолчанию.

Конструктор по умолчанию

Если вы не объявляете *явно* ни одного конструктора, то компилятор создаст для класса свой конструктор по умолчанию, который:

1. *Ответ на уд*: ничего не делает (инициализирует чем попало).
2. *Ответ на хор*: инициализирует все поля их конструкторами по умолчанию.
3. *Ответ на отл*: инициализирует все поля-классы конструкторами по умолчанию, а для полей-примитивных типов ничего не делает.

```
class A { ... };

class B {
  int x;
  A a;
  // B() <-- компилятор создаст за вас, если нет упоминания конструкторов
};

B b;  // b.x не проинициализирован, а для b.a вызван к-р по умолчанию
```

Конструктор по умолчанию

Но, если в классе вы *явно объявили хотя бы один* конструктор, то никакого неявного конструктора по умолчанию компилятор создавать не будет.

Конструктор по умолчанию

Конструктор по умолчанию предоставляемый компилятором эквивалентен конструктору с пустым телом:

```
class A { ... };

class B {
  int x;
  A a;

  B() {}  // <=> B() : a() {}
};
```

Напомню, это потому, что список инициализации выполняется **в любом случае**.

Конструктор по умолчанию

Компилятор также может отказаться генерировать свой конструктор по умолчанию, если он не понимает как это сделать.

```
struct A {
private:
  A() {}
};

struct B {
  A a;           // Oops: приватный конструктор A()
  const int b;  // Oops: как проинициализировать константу?
  int& c;        // Oops: как проинициализировать ссылку?
};

B b;  // Compilation error
```

```
error: use of deleted function 'B::B()'
```

`= default`

- Допустим, в вашем классе уже есть какой-то конструктор, а писать конструктор по умолчанию самостоятельно не хочется.
- Можно явно попросить компилятор предоставить свою версию конструктора по умолчанию с помощью `= default`.
- Такой конструктор эквивалентен конструктору с пустым телом.

```
struct B {
  B(int x, int y) {}
  B() = default;  // эквивалентно B() {}
};
```

Виды конструкторов: конструктор копирования

Конструктор копирования

- *Конструктор копирования* - конструктор, который создает объект с помощью другого объекта того же типа путем его копирования.
- Как следует из определения, единственный аргумент - объект того же типа.
- Однако тривиальная передача по значению обречена на провал:

```
Stack::Stack(Stack s) : buffer_(new int[kCapacity]), size_(s.size_) {
  for (size_t i = 0; i < size_; ++i) {
    buffer_[i] = s.buffer_[i];
  }
}

Stack a;
Stack b(a);  // Бесконечная рекурсия!
```

```
error: invalid constructor
```

Конструктор копирования

- Решение: давайте передавать аргумент по *константной* ссылке.

```
Stack::Stack(const Stack& other) { ... }

Stack a;
Stack b(a);  // Ok
```

- Константность нужна, чтобы работал следующий код:

```
const Stack a;
// ...
Stack b(a);          // Ok
Stack c(Stack());  // Ok даже до C++17
```

- Кроме того, так мы точно случайно не испортим исходный объект.

Конструктор копирования

Если вы не объявляете своего конструктора копирования (и конструктора перемещения), то компилятор создаст для класса свой конструктор копирования, который:

1. *Ответ на уд*: побитово копирует все поля
2. *Ответ на хор*: вызывает конструктор копирования для каждого из полей
3. *Ответ на отл*: поля-классы инициализируются конструкторами копирования, а поля-базовые типы копируются побитово

```
class A { ... };

class B {
  A a;
  int x;
  // v подарок от компилятора
  // B(const B& other) : a(other.a), x(other.x) {}
};
```

Конструктор копирования

Компилятор откажется это делать, если в классе есть поля, которые нельзя скопировать.

```
struct A {
  A() = default;

private:
  A(const A&);
};

struct B {
  A a;
};

B first;
B second(first);  // Compilation error
```

```
error: use of deleted function 'B::B(const B&)'
```

`= default`

- Аналогично конструктору по умолчанию можно *явно* попросить компилятор создать свою версию конструктора копирования с помощью `= default`.

```
struct B {
  B() = default;
  B(const B&) = default;
};

B first;
B second(first); // Ok
```

- Хотя это не является обязательным - достаточно просто не написать свой конструктор копирования, чтобы компилятор сгенерировал свой.
- В каких ситуациях стоит писать свой конструктор копирования, а в каких стоит довериться компилятору?

Замечание

Для создания дефолтного конструктора копирования **недостаточно** определить его с пустыми фигурными скобками:

```
class A { ... };

struct B {
  int x;
  A a;

  B() = default;  // <=> B() : a() {}
  B() {};         // <=> B() : a() {}

  B(const B&) = default;  // <=> B(const B&) : x(other.x), a(other.a) {}
  B(const B&) {}          // <=> B(const B&) : a() {}
//^ !!!
};
```

Делегирующие конструкторы

Делегирующие конструкторы (C++11)

Часто так бывает, что конструкторы дублируют действия друг друга.

```
Stack::Stack(size_t size, const int* values)
  : buffer_(new int[kCapacity])
  , size_(size) {

  for (size_t i = 0; i < size; ++i) {
    buffer_[i] = values[i];
  }
}

Stack::Stack(const Stack& other)  // Дублирование кода!
  : buffer_(new int[kCapacity])
  , size_(other.size_) {

  for (size_t i = 0; i < size_; ++i) {
    buffer_[i] = other.buffer_[i];
  }
}
```

Делегирующие конструкторы (C++11)

Хотелось бы иметь возможность вызывать конструктор в другом конструкторе.

```
Stack::Stack(size_t size, const int* values)
  : buffer_(new int[kCapacity])
  , size_(size) {

  for (size_t i = 0; i < size; ++i) {
    buffer_[i] = values[i];
  }
}

Stack::Stack(const Stack& other) {
  Stack(other.size_, other.buffer_);  // Создание временного объекта
}
```

Проблема в том, что здесь происходит создание временного объекта, а не вызов конструктора.

Делегирующие конструкторы (C++11)

Решение: использовать делегирование конструктора

```
Stack::Stack(size_t size, const int* values)
  : buffer_(new int[kCapacity])
  , size_(size) {

  for (size_t i = 0; i < size; ++i) {
    buffer_[i] = values[i];
  }
}

Stack::Stack(const Stack& other) : Stack(other.size_, other.buffer_) {
}
```

Если применено делегирование, то инициализация полностью на совести вызванного конструктора, отдельные поля проинициализировать не получится.

Деструктор

Деструктор

- *Деструктор* - особый метод класса, который вызывается при завершении времени жизни объекта.
- Этот метод не имеет возвращаемого значения и аргументов.
- Его имя = `~<ИмяКласса>`.
- Деструктор вызывается *неявно* при уничтожении любого объекта, однако может быть вызван и явно (как метод). Но так делать не стоит (как правило, приводит к UB).

```
Stack::~Stack() {
  delete[] buffer_;
}

Stack stack;
stack.Push(1);
// Ok, утечек памяти нет
```

Деструктор

- Если вы не пишите своего деструктора, то компилятор создаст для класса свой, который:

1. *Ответ на уд*: ничего не делает.
2. *Ответ на хор*: вызывает деструкторы полей.
3. *Ответ на отл*: вызывает деструкторы для полей-классов, а для полей-базовых типов ничего не делает.

- Если невозможно вызвать деструктор у какого-либо поля, то компилятор откажется создавать свой деструктор.
- Чтобы явно указать намерение использовать предоставленный компилятором деструктор, можно использовать `= default`.

Деструктор

- Если деструктор недоступен, то вы не сможете создавать объекты на стеке и в статической области памяти.

```
class A {
  ~A() = default;  // приватный конструктор
};

A a;
```

```
error: 'A::~A()' is private within this context
```

- Но можно создавать объекты в динамической области. Правда, в этом случае придется управлять памятью на низком уровне

Что делать в деструкторе?

- Если при уничтожении объекта не требуется выполнения нетривиальных действий, то ничего (лучше предоставить работу компилятору).
- Если уничтожение объекта требует освобождения выделенной памяти, закрытия файлов, логирования и т.д., прописывайте эти действия в деструкторе (компилятор не догадается самостоятельно вызвать `delete`!).
- Важно помнить, что, хотите вы того или нет, у каждого поля при выходе из тела деструктора вызовется свой деструктор, поэтому вручную уничтожать поля не нужно.

Что делать в деструкторе?

```
Stack::~Stack() {
  delete[] buffer_;  // Ok
  size_ = 0;         // Ok, но зачем?
}

struct B {
  Stack s;

  ~B() {
    s.~Stack();  // UB: не надо так
  }  // <-- деструктор Stack здесь вызовется снова
};
```

RAII

Конструкторы и деструкторы позволяют реализовать важнейшую идиому языка C++ - **RAII** *(Resource Acquisition Is Initialization/Захват Ресурса - это Инициализация)*

Идея в том, чтобы выделение и освобождение ресурса происходило автоматически (в конструкторе и деструкторе соответственно)

```
auto ptr = new int(10);  // не безопасно! Можно забыть delete
```

```
// RAII
class IntPtr {
  int* ptr;

public:
  explicit IntPtr(int value) : ptr(new int(value)) {}
  ~IntPtr() { delete ptr; }
    // ...
};
```

Порядок вызова конструкторов/деструкторов

Порядок вызова конструкторов/деструкторов

Стековые объекты создаются в порядке объявления, а уничтожаются в обратном

```
struct A {
  A()  { std::cout << "A() "; }
  ~A() { std::cout << "~A() "; }
};

struct B {
  B()  { std::cout << "B() "; }
  ~B() { std::cout << "~B() "; }
};

int main() {
  A a;
  B b;
}
```

```
A() B() ~B() ~A()
```

Правило трех

Правило трех

- В C++ существует правило приличия, которое носит название *правило трех*:

"Если класс требует реализации пользовательского деструктора либо конструктора копирования, либо операции присваивания, то требуется реализовать все эти три сущности"

- Несмотря на то, что это правило не является правилом языка (ошибки компиляции нарушение этого правила не провоцирует), важно следовать этому правилу для корректной работы ваших классов
- Далее в курсе это правило эволюционирует в *правило пяти* и *правило ноля*

Правило трех

```
class IntPtr {
  int* ptr;

public:
  IntPtr(int value) : ptr(new int(value)) {}

  IntPtr(const IntPtr& other) : ptr(new int(*other.ptr)) {}

  IntPtr& operator=(const IntPtr& other) { *ptr = *other.ptr; }

  ~IntPtr() { delete ptr; }
};
```

Что произойдет, если не реализовать хотя бы один из методов?

```
= delete
```

`= delete` (C++11)

Начиная с C++11, можно объявлять функции "удаленными". Такие функции нельзя вызывать, а также нельзя получать указатель на них.

```
void f(int);
void f(double) = delete;

template <class T> void g(T);
template <> void g(int) = delete;
// ...

f(1);    // Ok
f(1.0);  // CE
g(1);    // CE
g(1.0);  // Ok
```

```
error: call to deleted function 'f'
error: call to deleted function 'g'
```

`= delete` (C++11)

- Как правило, эта возможность используется для запрета генерации некоторых методов (например, для запрета копирования).

```
struct C {
  C(const C&) = delete;  // теперь к-р копирования не может быть вызван
    // ...
};
```

- Альтернативно, можно объявить метод приватным.

```
struct C {
private:
  C(const C&);
    // ...
};
```

Чем первый способ лучше второго?

Резюме

- Конструктор - специальный метод, инициализирующий объекты класса.
- Для эффективной инициализации полей используйте списки инициализации, а делегирование конструкторов поможет избежать дублирования кода.
- Деструктор - специальный метод, который вызывается при удалении объекта.
- Соблюдайте УК, ГК и правило трех.
- Используйте `= default` и `= delete`.