МГТУ
им. Н.Э. Баумана
ТЕХНОПАРК
Mail.Ru Group

Углубленное программирование на языке C++

Алексей Петров

Лекция №3. Основные вопросы ООП на языке C++

1. Инкапсуляция и ответственность класса. Принципы SRP, OCP. Идиома RAII.
2. Праводопустимые выражения. Конструкторы (операции) переноса и иные расширения объектной модели в C++11.
3. Инкапсуляция и вопросы производительности.
4. **Постановка задач к практикуму №3.**

Рекомендуемая литература: модуль №2 (1 / 2)

- Дейтел Х., Дейтел П. Как программировать на C++. — Бином-Пресс, 2009. — 800 с.
- Липпман С., Лажойе Ж. Язык программирования C++. Вводный курс. — Невский Диалект, ДМК Пресс. — 1104 с.
- Липпман С., Лажойе Ж., Му Б. Язык программирования C++. Вводный курс. — Вильямс, 2007. — 4-е изд. — 896 с.
- Прата С. Язык программирования C++. Лекции и упражнения. — Вильямс, 2012. — 6-е изд. — 1248 с.: ил.
- Саттер Г. Новые сложные задачи на C++. — Вильямс, 2005. — 272 с.
- Саттер Г. Решение сложных задач на C++. — Вильямс, 2008. — 400 с.

Рекомендуемая литература: модуль №2 (2 / 2)

- Саттер Г., Александреску А. Стандарты программирования на C++. — Вильямс, 2008. — 224 с.
- Страуструп Б. Программирование. Принципы и практика использования C++. — Вильямс, 2011. — 1248 с.
- Страуструп Б. Язык программирования C++. — Бином, 2011. — 1136 с.
- Шилдт Г. C++: базовый курс. — Вильямс, 2008. — 624 с.
- Шилдт Г. C++. Методики программирования Шилдта. — Вильямс, 2009. — 480 с.
- Шилдт Г. Полный справочник по C++. — Вильямс, 2007. — 800 с.
- Abrahams, D., Gurtovoy, A. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond* (Addison Wesley Professional, 2004).

Инкапсуляция — базовый принцип ООП

Инкапсуляция, или сокрытие реализации, является фундаментом объектного подхода к разработке ПО.

- Следуя данному подходу, программист рассматривает задачу в терминах предметной области, а создаваемый им продукт видит как **совокупность абстрактных сущностей — классов** (в свою очередь формально являющихся пользовательскими типами).
- Инкапсуляция **предотвращает прямой доступ** к внутреннему представлению класса из других классов и функций программы.
- Без нее теряют смысл остальные основополагающие принципы объектно-ориентированного программирования (ООП): наследование и полиморфизм. Сущность инкапсуляции можно отразить формулой:

Открытый интерфейс + скрытая реализация

Класс: в узком или широком смысле?

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

Принцип инкапсуляции распространяется не только **на классы** (class), но и **на структуры** (struct), а также **объединения** (union). Это связано с расширительным толкованием понятия «класс» в языке C++, трактуемом как в узком, так и широком смысле:

- **класс в узком смысле** — *одноименный* составной пользовательский тип данных, являющийся контейнером для данных и алгоритмов их обработки. Вводится в текст программы определением типа со спецификатором class;
- **класс в широком смысле** — *любой* составной пользовательский тип данных, агрегирующий данные и алгоритмы их обработки. Вводится в текст программы определением типа с одним из спецификаторов struct, union или class.

Каждое определение класса вводит **новый тип данных.** Тело класса определяет **полный перечень его членов**, который не может быть расширен после закрытия тела.

Указатель this

Указатель this — **неявно определяемый константный указатель** на объект класса, через который происходит вызов соответствующего нестатического метода.

Для неконстантных методов класса T имеет тип T *const, для константных — имеет тип const T *const, для неустойчивых — volatile T *const. Указатель this допускает разыменование (*this).

Применение this внутри методов допустимо, но чаще всего излишне. Исключение составляют две ситуации:

- сравнение адресов объектов:

```
if (this != someObj) /* ... */
```

- оператор return:

```
return *this;
```

Класс как область видимости

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

Класс — наряду с блоком, функцией и пространством имен — **является конструкцией** C++, которая **вводит** в состав программы одноименную **область видимости**. (Строго говоря, область видимости вводит определение класса, а именно его тело.)

- Все члены класса видны в нем самом с момента своего объявления. Порядок объявления членов класса важен: нельзя ссылаться на члены, которые предстоит объявить позднее. Исключение составляет разрешение имен в определениях встроенных методов, а также имен (статических членов), используемых как аргументы по умолчанию.

В области видимости класса находится не только его тело, но и внешние определения его членов: методов и статических атрибутов.

Конструкторы и деструкторы (1 / 2)

Конструктор — метод класса, автоматически применяемый к каждому экземпляру (объекту) класса перед первым использованием (в случае динамического выделения памяти — после успешного выполнения операции new).

Освобождение ресурсов, захваченных в конструкторе класса либо на протяжении времени жизни соответствующего экземпляра, осуществляет **деструктор**.

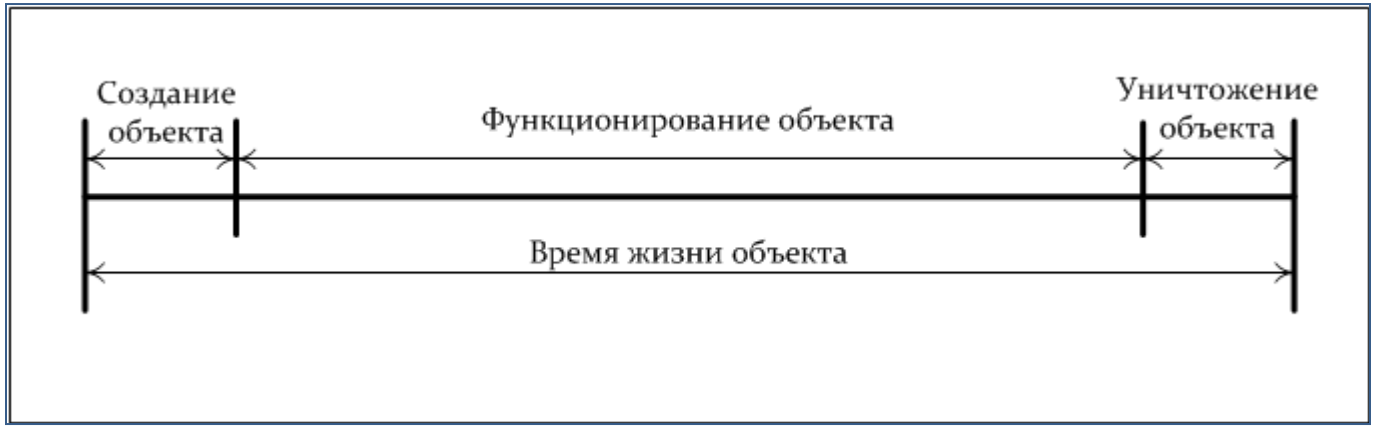
В связи с принятым по умолчанию почленным порядком инициализации и копирования объектов класса в большинстве случаев возникает необходимость в реализации, — наряду с конструктором по умолчанию, — **конструктора копирования** и перегруженной **операции-функции присваивания** operator=.

Конструкторы и деструкторы (2 / 2)

Выполнение любого конструктора состоит из двух фаз:

- фаза явной (неявной) инициализации (обработка списка инициализации);
- фаза вычислений (исполнение тела конструктора).

Конструктор **не может определяться** со спецификаторами const и volatile. Константность и неустойчивость объекта устанавливается по завершении работы конструктора и снимается перед вызовом деструктора.



Инициализация без конструктора (1 / 2)

Класс, все члены которого открыты, может задействовать механизм **явной позиционной инициализации**, ассоциирующий значения в списке инициализации с членами данных в соответствии с их порядком.

```
class Test
{
public:
        int    int_prm;
        double dbl_prm;
        string str_prm;
};

// …
Test test = { 1, -3.14, "dictum factum" };
```

Инициализация без конструктора (2 / 2)

Преимуществами такой техники выступают:

- скорость и эффективность, особо значимые при выполнении во время запуска программы (для глобальных объектов).

Недостатками инициализации без конструктора являются:

- пригодность только для классов, члены которых открыты;
- отсутствие поддержки инкапсуляции и абстрактных типов;
- требование предельной точности и аккуратности в применении.

Конструкторы по умолчанию (1 / 2)

Конструктор по умолчанию **не требует задания значений** его параметров, хотя таковые могут присутствовать в сигнатуре.

```
class Test
{
public:
        Test(int ipr = 0, double dpr = 0.0);
        /* … */
};
```

Наличие формальных параметров в конструкторе по умолчанию позволяет **сократить общее число конструкторов** и объем исходного кода.

Конструкторы по умолчанию (2 / 2)

Если в классе определен хотя бы один конструктор с параметрами, то при использовании класса со стандартными контейнерами и динамическими массивами экземпляров конструктор по умолчанию **обязателен**.

```
Test *tests = new Test[TEST_PLAN_SIZE];
```

Если конструктор по умолчанию **не определен**, но существует хотя бы один конструктор с параметрами, в определении объектов должны присутствовать аргументы. Если ни одного конструктора не определено, объект класса не инициализируется (память под статическими объектами по общим правилам обнуляется).

Конструкторы с параметрами: пример


```
class Test
{
public:
        Test(int prm) : _prm (prm) {}
private:
        int     _prm;
};

// все вызовы конструктора допустимы и эквивалентны
Test    test1(10),
        test2 = Test(10),
        test3 = 10;    // для одного аргумента
```

Массивы объектов: пример

```
// массивы объектов класса определяются
// аналогично массивам объектов базовых типов

// для конструктора с одним аргументом
Test testplan1[] = { 10, -5, 0, 127 };

// для конструктора с несколькими аргументами
Test testplan2[5] = {
       Test(10, 0.1),
       Test(-5, -3.6),
       Test(0, 0.0),
       Test() // если есть конструктор по умолчанию
};
```

Закрытые и защищенные конструкторы

Описание конструктора класса как **защищенного** или **закрытого** дает возможность ограничить или полностью запретить отдельные способы создания объектов класса.

В большинстве случаев закрытые и защищенные конструкторы используются для:

- **предотвращения копирования** одного объекта в другой;
- указания на то, что конструктор **должен вызываться** только **для создания подобъектов** базового класса в объекте производного класса, а не создания объектов, непосредственно доступных в коде программы.

Почленная инициализация и присваивание (1 / 2)

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

Почленная инициализация по умолчанию — механизм инициализации одного объекта класса другим объектом того же класса, который активизируется независимо от наличия в определении класса явного конструктора.

Почленная инициализация по умолчанию происходит в следующих ситуациях:

- явная инициализация одного объекта другим;
- передача объекта класса в качестве аргумента функции;
- передача объекта класса в качестве возвращаемого функцией значения;
- определение непустого стандартного последовательного контейнера;
- вставка объекта класса в стандартный контейнер.

Почленная инициализация и присваивание (2 / 2)

Почленная инициализация по умолчанию **подавляется** при наличии в определении класса конструктора копирования.

Запрет почленной инициализации по умолчанию осуществляется одним из следующих способов:

- описание закрытого конструктора копирования (не действует для методов класса и дружественных объектов);
- описание конструктора копирования без его определения (действует всюду).

Почленное присваивание по умолчанию — механизм присваивания одному объекту класса значения другого объекта того же класса, отличный от почленной инициализации по умолчанию использованием копирующей операции-функции присваивания вместо конструктора копирования.

Конструкторы копирования

Конструктор копирования принимает в качестве единственного параметра **константную ссылку** на существующий объект класса.

В случае отсутствия явного конструктора копирования в определении класса производится почленная инициализация объекта по умолчанию.

```
class Test
{
        /* … */
        Test(const Test &other);
        /* … */
};
```

Конструкторы и операции преобразования

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

Конструкторы преобразования служат для построения объектов класса по одному или нескольким значениям иных типов.

Операции преобразования позволяют преобразовывать содержимое объектов класса к требуемым типам данных.

```
class Test
{       // конструкторы преобразования
        Test(const char *);
        Test(const string &);
        // операции преобразования
        operator int    () { return int_prm; }
        operator double () { return dbl_prm; }
        /* … */
};
```

Деструкторы. Виртуальные деструкторы (1 / 2)

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

Деструктор — не принимающий параметров и не возвращающий результат метод класса, автоматически вызываемый при выходе объекта из области видимости и применении к указателю на объект класса операции delete.

```
class Test
{
        /* … */
        virtual ~Test();
};
```

Примечание: деструктор не вызывается при выходе из области видимости ссылки или указателя на объект.

Деструкторы.
Виртуальные деструкторы (2 / 2)

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

Типичные задачи деструктора:

- сброс содержимого программных буферов в долговременные хранилища;
- освобождение (возврат) системных ресурсов, главным образом — оперативной памяти;
- закрытие файлов или устройств;
- снятие блокировок, останов таймеров и т.д.

Для обеспечения корректного освобождения ресурсов объектами производных классов деструкторы в иерархиях, как правило, определяют как **виртуальные**.

Идиома RAII

Закрепление за конструкторами функции захвата, выделения, блокировки или инициализации ресурсов, а **за деструкторами — функции** их возврата, освобождения и снятия установленных блокировок:

- позволяет **безопасно обрабатывать ошибки и исключения**;
- составляет суть одной из важнейших идиом ОО-программирования RAII (англ. Resource Acquisition Is Initialization — «получение ресурса есть инициализация»).

Работа идиомы RAII в языке C++ основана, главным образом, на **гарантированном вызове деструкторов автоматических переменных**, являющихся экземплярами классов, при выходе из соответствующих областей видимости.

Явный вызов деструкторов

Потребность в явном вызове деструктора обычно связана с необходимостью **уничтожить** динамически размещенный объект **без освобождения памяти**.

```
char *buf = new char[sizeof(Test)];
// "размещающий" вариант new
Test *ptc = new (buf) Test(100);

/* … */
ptc->~Test();       // вызов 1
Test *ptc = new (buf) Test(200);

/* … */
ptc->~Test(); // вызов 2
delete [] buf;
```

Список инициализации в конструкторе

Выполнение любого конструктора состоит из двух фаз:

- фаза явной (неявной) инициализации (обработка списка инициализации) — предполагает **начальную инициализацию** членов данных;
- фаза вычислений (исполнение тела конструктора) — предполагает **присваивание значений** (в предварительно инициализированных областях памяти).

Присваивание значений членам данных – объектам классов в теле конструктора **неэффективно** ввиду ранее произведенной инициализации по умолчанию. Присваивание значений членам данных, представляющих базовые типы, **по эффективности равнозначно** инициализации.

К началу исполнения тела конструктора все **константные члены и члены-ссылки** должны быть инициализированы.

Семантика переноса (C++11)

Введение в C++11 семантики переноса (англ. move semantics) обогащает язык возможностями более **тонкого и эффективного управления памятью данных**, устраняющего копирование объектов там, где оно нецелесообразно.

Технически семантика переноса реализуется при помощи **ссылок на праводопустимые выражения** и **конструкторов переноса**.

Конструкторы переноса не создают точную копию своего параметра, а перенастраивают параметр так, чтобы права владения соответствующей областью памяти были переданы вновь создаваемому объекту («заимствованы» последним).

Аналогично работают **операции присваивания с переносом**.

Конструктор переноса: пример (1 / 2)


```
class Alpha {
public:
        Alpha();
        Alpha(const Alpha &a); // конструктор копирования
        Alpha(Alpha &&a);      // конструктор переноса
        ~Alpha();
private:
        size_t sz;
        double *d;
};

Alpha::Alpha() : sz(0), d(0) { }

Alpha::~Alpha() {
        delete [] d;
}
```

Конструктор переноса: пример (2 / 2)

```
// конструктор копирования
Alpha::Alpha(const Alpha &a) : sz(a.sz)
{
        d = new double[sz];
        /* … */
        for(size_t i = 0; i < sz; i++)
                d[i] = a.d[i];
}

// конструктор переноса
Alpha::Alpha(Alpha &&a) : sz(a.sz)
{
        d    = a.d;
        // перенастройка параметра
        a.d  = nullptr; // C++11
        a.sz = 0;
}
```

Принципы S.O.L.I.D.: начало

Принципы S.O.L.I.D. — устоявшееся обозначение «первой пятерки» принципов объектно-ориентированного программирования и дизайна, сформулированных главным редактором *C++ Report* Р. Мартином (Robert Martin) в начале 2000-х гг.

В число принципов S.O.L.I.D., обобщающих классические результаты 1980 – 1990-х гг., входят:

- Принцип единственной ответственности [Р. Мартин];
- Принцип открытости / закрытости [Б. Мейер (Bertrand Meyer)];
- Принцип подстановки Лисков [Б. Лисков (Barbara Liskov) — Ж. Уинг (Jeannette Wing)];
- Принцип разделения интерфейсов [Р. Мартин];
- Принцип инверсии зависимостей [Р. Мартин].

Принципы SRP и OCP

Принцип единственной ответственности (англ. Single Responsibility Principle, SRP) требует:

Любой класс должен иметь одну и только одну зону ответственности

Принцип открытости / закрытости (англ. Open / Closed Principle, OCP) гласит:

Программные элементы должны быть открыты для расширения, но закрыты для изменения

Практикум №3

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

Постановка задачи

- Сформировать команду (выполнено!).
- Предложить собственную тему проекта (см. блог дисциплины).
- Построить концептуальную UML-модель предметной области проекта и детализировать состав основных классов.
- **Цель** — спроектировать полиморфную иерархию из трех или более классов с множественным наследованием, семантика и функциональная нагрузка которых определяются темой проекта.

Спасибо за внимание

Алексей Петров