ORES

Научно-практический электронный журнал
Оригинальные исследования (ОРИС), *ISSN 2222-0402*
www.ores.su
original-research.ru

УДК 528.067 + 004.4'23

ЭФФЕКТИВНОЕ ИСПОЛЬЗОВАНИЕ GDAL В C++: ОТ РУЧНОГО УПРАВЛЕНИЯ РЕСУРСАМИ К RAII

Лукин Иван Петрович,
МИРЭА - Российский технологический университет (Москва, Россия)
E-mail: doggerr111@gmail.com

Аннотация

В статье представлен подход к созданию безопасных обёрток (wrappers) над GDAL-объектами с помощью идиомы RAII и умных указателей, обеспечивающий автоматическое освобождение ресурсов и повышающий надёжность и читаемость кода.

Ключевые слова: API, C++, GDAL, RAII, ГИС, надёжность, пространственные данные, ресурсы, управление

EFFICIENT USE OF GDAL IN C++: FROM MANUAL RESOURCE MANAGEMENT TO RAII

Lukin Ivan Petrovich,
MIREA - Russian Technological University (Moscow, Russia)

ABSTRACT

The article presents an approach to creating secure wrappers over GDAL objects using the RAII idiom and smart pointers, which provides automatic resource release and increases the reliability and readability of the code.

Keywords: API, C++, GDAL, RAII, GIS, reliability, spatial data, resources, management

Введение

Открытая библиотека преобразования растровых и векторных данных GDAL (Geospatial Data Abstraction Library) — библиотека-транслятор для геопространственных форматов данных, выпущенная под лицензией MIT компанией Open Source Geospatial Foundation. Она предоставляет единую абстрактную модель данных для всех поддерживаемых форматов, а также набор утилит командной строки для трансляции и обработки данных [407].

GDAL — незаменимый инструмент при работе с пространственными данными, применяемый как в проприетарном, так и в открытом программном обеспечении. Большинство современных геоинформационных систем (ГИС) на том или ином уровне

используют эту библиотеку. Среди открытых проектов можно выделить QGIS — одну из самых популярных настольных ГИС, GRASS GIS, SAGA GIS, а также серверные решения, такие как GeoServer и MapServer. В мире проприетарного программного обеспечения GDAL используется в ArcGIS Pro от ESRI, MapInfo Professional, FME и многих других [407].

GDAL изначально поддерживала только растровые данные. По мере роста популярности проект расширился и включил векторные данные через библиотеку-компаньон OGR. К 2008 году GDAL и OGR уже развивались в рамках единой кодовой базы, однако окончательная унификация программного интерфейса (API) произошла с выходом версии 2.0 в 2015 году, что упростило экосистему и укрепило GDAL как главный инструмент для работы с геопространственными данными.

Сейчас GDAL обеспечивает унифицированный доступ к десяткам растровых и векторных форматов: Shapefile, GeoPackage, GeoTIFF, PostGIS и другим. Благодаря этому ГИС-приложения могут читать и записывать данные без привязки к конкретному формату. Библиотека также предоставляет утилиты для перепроецирования, конвертации и пространственной обработки.

Несмотря на то, что современный GDAL написан на C++, его API сохраняет C-стиль управления ресурсами. Это вынуждает разработчика вручную управлять временем жизни объектов — освобождать память и закрывать дескрипторы, что усложняет код и повышает риск утечек.

Цель данной статьи — показать, как с помощью идиомы RAII и умных указателей std::unique_ptr можно создать безопасные обёртки над GDAL-объектами, автоматизирующие освобождение ресурсов и делающие код более надёжным и читаемым.

C-стиль GDAL и его преодоление с помощью RAII

Интерфейс в стиле C обеспечивает стабильный бинарный интерфейс приложения (ABI), что упрощает связывание с другими языками через инструменты типа SWIG (Simplified Wrapper and Interface Generator), который позволяет связывать программы и библиотеки на языках C/C++ с интерпретируемыми языками программирования (Python, Java, C# и т.д.). Из-за необходимости поддерживать C-совместимый интерфейс, многие возможности современного C++, такие как умные указатели, контейнеры стандартной библиотеки (std::vector, std::string), идиома RAII, а также шаблоны и метапрограммирование, практически не используются в публичном API GDAL.

Это обстоятельство создаёт определённые неудобства при разработке геоинформационных приложений на C++. Программист вынужден вручную управлять ресурсами, что повышает сложность кода и риск ошибок. C-стиль GDAL проявляется и в других аспектах API: непрозрачные указатели (дескрипторы вроде GDALDatasetH), передача метаданных через char**, коды ошибок вместо исключений C++, а также монолитные заголовочные файлы (например, gdal_priv.h).

Особенно остро эта проблема заметна при работе с объектами GDAL, которые требуют ручного освобождения ресурсов. Вместо автоматического вызова деструкторов программист вынужден явно управлять временем жизни каждого такого объекта.

Чтобы лучше понять проблему, рассмотрим типичный пример чтения векторного слоя из файла с помощью GDAL, используя C++ API [407]:

```
void readVectorLayer(const char* path)
{
  GDALAllRegister();
  GDALDataset* ds = static_cast<GDALDataset*>(GDALOpenEx(path, GDAL_OF_VECTOR, nullptr, nullptr, nullptr));
  if (!ds) return;
```

```
    OGRLayer* layer = ds->GetLayer(0);
    layer->ResetReading();
    OGRFeature* feature;
    while ((feature = layer->GetNextFeature()) != nullptr) {
        // обработка feature
        // ... (здесь может быть исключение или ранний return)
        OGRFeature::DestroyFeature(feature); // нужно не забыть!
    }
    GDALClose(ds); // нужно не забыть!
}
```

При использовании GDAL на программиста возлагается ответственность за явное освобождение ресурсов. В приведённом примере чтение векторного слоя начинается с вызова GDALOpenEx() – функции, которая возвращает указатель на GDALDataset. Этот объект содержит всю информацию о наборах пространственных данных — растровых каналах или векторных слоях — в зависимости от типа открытого источника (файл, база данных, веб-сервис и т.д. Затем метод GetLayer() предоставляет доступ к векторному слою, после чего в цикле while происходит чтение всех векторных объектов этого слоя.

При завершении работы с датасетом необходимо явно вызвать GDALClose(). Если этого не сделать, память, выделенная под датасет, останется занятой до завершения программы, что приведёт к утечке. Кроме того, программист обязан вручную уничтожать каждую копию векторного объекта, полученную через GetNextFeature(), с помощью OGRFeature::DestroyFeature(). Всё это создаёт благоприятную почву для утечек памяти.

Усложним пример: предположим, что в цикле while выполняются дополнительные проверки геометрии объектов.

```
OGRFeature* feature;
    while ((feature = layer->GetNextFeature()) != nullptr)
    {
        OGRGeometry* geom = feature->GetGeometryRef();
        // Пропускаем объекты без геометрии
        if (!geom) {
            // Забыли DestroyFeature(feature) перед continue!
            continue;
        }
        // Получаем bounding box
        OGREnvelope envelope;
        geom->getEnvelope(&envelope);
        // Фильтр по X
        if (envelope.MinX < minX || envelope.MaxX > maxX) {
            // Забыли DestroyFeature(feature) перед continue!
            continue;
        }
    }
GDALClose(ds); // нужно не забыть!
```

В приведённом фрагменте мы пытаемся получить геометрию векторного объекта. Условный оператор if (!geom) проверяет, обладает ли объект геометрией — если нет, объект пропускается. Далее мы получаем ограничивающий прямоугольник (bounding box) геометрии и применяем простейший фильтр по координате X. Если какое-либо из условий

не выполняется, мы должны перейти к следующему объекту, но перед этим вручную вызвать OGRFeature::DestroyFeature() для текущего.

Уже в этом простом примере легко забыть про освобождение ресурса. А что, если таких условий будет не два, а десять? С увеличением сложности логики вероятность ошибки возрастает многократно. Разработчик вынужден постоянно следить за тем, чтобы каждый путь выполнения завершался вызовом DestroyFeature. Это не просто неудобно — это опасно. Рано или поздно такой стиль программирования гарантированно приведёт к утечке памяти.

Но представим другую ситуацию. Что если между вызовами GDALOpenEx() и GDALClose() возникнет исключение? Например, при обработке геометрии или при работе с атрибутами. В этом случае начнется раскрутка стека, в ходе которой уничтожатся локальные объекты, но не будет выполнено закрытие датасета и уничтожение копий объектов OGRFeature. Программа потеряет возможность освободить ресурсы, и память будет утекать. Это делает код не только громоздким, но и принципиально ненадёжным с точки зрения обработки исключительных ситуаций. Рассмотрим конкретный пример:

```
void dangerous(const char* path)
{
   GDALAllRegister();
   GDALDataset* ds = (GDALDataset*)GDALOpen(path, GA_ReadOnly);
   if (!ds) return;
   OGRLayer* layer = ds->GetLayer(0);
   OGRFeature* feature = layer->GetNextFeature();
   // Здесь может быть исключение (например, при работе с геометрией)
   processFeature(feature); //Если исключение — feature не уничтожен, ds не
                            закрыт
   OGRFeature::DestroyFeature(feature);
   GDALClose(ds);
}
```

Предположим, функция processFeature() выполняет какие-то действия по обработке векторного объекта и не объявлена как noexcept, тогда в её коде может встретиться, например, следующее:

```
void processFeature(OGRFeature* feature)
{
   auto* geom = feature->GetGeometryRef();
   if (!geom)
      throw std::runtime_error("nullptr feature geometry!"); //Исключение!
};
```

В случае возникновения этого исключения начнётся раскрутка стека. Для локальных объектов будут вызваны деструкторы, однако объекты, размещённые в динамической памяти (GDALDataset* ds, OGRFeature* feature), останутся неудалёнными. Ответственность за их освобождение лежит на программисте, но из-за исключения управление не дойдёт до вызовов GDALClose() и OGRFeature::DestroyFeature(). В результате произойдёт утечка памяти.

И это лишь один пример. В действительности ручного освобождения могут потребовать многие другие объекты GDAL: геометрии, системы координат, описания атрибутивных полей и т.д. Чем больше таких объектов, тем выше риск ошибки — особенно в нетривиальных сценариях с исключениями и множественными точками выхода.

Таким образом, ручное управление ресурсами в GDAL — это не просто вопрос дисциплины, а потенциальный источник трудноуловимых ошибок, которые могут проявиться только при определённых сценариях выполнения программы.

Предлагаемое решение

Решением этой проблемы является применение идиомы RAII (Resource Acquisition Is Initialization). Её суть состоит в следующем: захват ресурса происходит в конструкторе объекта, а освобождение — в деструкторе. Это гарантирует, что ресурс будет освобождён при любом выходе из области видимости: при нормальном завершении, при исключении или при раннем возврате.

В стандартной библиотеке C++ RAII реализован в умных указателях (std::unique_ptr, std::shared_ptr). Пользователю не нужно заботиться о ручном освобождении ресурса — деструктор сделает это автоматически. Применим этот подход к работе с GDAL: создадим обёртку над GDALDataset*, которая закроет датасет в своём деструкторе, и обёртку над OGRFeature*, которая уничтожит векторный объект. Тогда даже при исключении или преждевременном выходе из функции ресурсы будут освобождены корректно.

Для создания такой обёртки необходимо определить пользовательский удалитель — функтор, который будет передан в умный указатель (или станет частью его типа в случае std::unique_ptr). Такие функторы инкапсулируют вызов правильной функции освобождения. Рассмотрим, как такие функторы могут выглядеть для GDALDataset и OGRFeature:

```
struct GdalDatasetDeleter { //Функтор-удалитель для GDALDataset*
   void operator()(GDALDataset* dataset) const {
      if (dataset)
         GDALClose(dataset);
   }
};
struct OgrFeatureDeleter { //Функтор-удалитель для OGRFeature*
   void operator()(OGRFeature* feature) const {
      if (feature)
         OGRFeature::DestroyFeature(feature)
   }
};
```

Представленные выше функторы принимают указатель на соответствующий объект GDAL и вызывают для него корректную функцию освобождения ресурсов. Для удобства использования определим алиасы (от англ. alias — псевдоним, альтернативное имя) с помощью ключевого слова using:

```
using GdalDatasetPtr = std::unique_ptr<GDALDataset, GdalDatasetDeleter>;
using OgrFeaturePtr = std::unique_ptr<OGRFeature, OgrFeatureDeleter>;
```

Теперь GdalDatasetPtr и OgrFeaturePtr ведут себя как обычные std::unique_ptr, но автоматически вызывают GDALClose() и OGRFeature::DestroyFeature() при своём уничтожении.

Для совместного владения можно использовать std::shared_ptr, передавая удалитель в конструктор. Однако в данной статье основное внимание уделяется std::unique_ptr как инструменту для исключительного владения ресурсами.

Для таких классов, как GDALDataset, имеет смысл написать фабричные функции, упрощающие создание умных указателей. Это инкапсулирует вызов GDALOpenEx и избавляет пользователя от необходимости явно приводить типы.

Таким образом, простейшая реализация фабричной функции для GDALDataset может выглядеть следующим образом:

```
inline GdalDatasetPtr createDataset(const std::string& str,
                                    unsigned int openFlags = GDAL_OF_VECTOR | GDAL_OF_READONLY)
{
  return GdalDatasetPtr(
    static_cast<GDALDataset*>(GDALOpenEx(str.c_str(), openFlags, nullptr, nullptr, nullptr))
  );
}
```

Функция createDataset принимает обязательный параметр str — строку с путём к файлу, параметрами подключения к базе данных или URL веб-сервиса. Второй параметр openFlags позволяет указать режим открытия: GDAL_OF_VECTOR, GDAL_OF_RASTER, GDAL_OF_READONLY, GDAL_OF_UPDATE и другие. Флаги можно комбинировать.

При необходимости можно создать более гибкую обёртку, передавая дополнительные аргументы (papszAllowedDrivers, papszOpenOptions, papszSiblingFiles) в GDALOpenEx напрямую или, например, через шаблоны с переменным числом аргументов, но в данной статье мы ограничимся базовыми параметрами — путём и флагами открытия.

Теперь применим написанные обёртки к функции чтения векторного слоя readVectorLayer(), которая была приведена ранее. Код становится значительно чище и безопаснее:

```
void readVectorLayer(const std::string& path)
{
  auto ds = createDataset(path, GDAL_OF_VECTOR); //Используем фабричную функцию, передавая путь к файлу и указывая флаг для открытия векторных данных
  if (!ds) return; //Проверяем, что датасет удалось открыть
  OGRLayer* layer = ds->GetLayer(0); //Обертка для OGRLayer не требуется, так как за время жизни таких объектов отвечает сам GDALDataset
  layer->ResetReading();
  while (auto feature = OgrFeaturePtr(layer->GetNextFeature())){
    // Работа с OgrFeature…
    // feature автоматически уничтожится при выходе из итерации
  }
  // ds автоматически закроется при выходе из области видимости функции
}
```

Использование std::unique_ptr с пользовательскими удалителями даёт несколько важных преимуществ:

Во-первых, объекты обёрток являются локальными. Это означает, что при выходе из области видимости (нормальном или при возникновении исключения) гарантированно вызываются их деструкторы, которые, в свою очередь, вызывают соответствующие удалители (GDALClose для датасета, OGRFeature::DestroyFeature для векторных объектов). Ресурсы освобождаются автоматически, без риска утечки.

Во-вторых, std::unique_ptr реализует семантику исключительного владения. В любой момент времени только один умный указатель может владеть ресурсом. Это делает код более прозрачным: глядя на сигнатуру функции, возвращающей GdalDatasetPtr, сразу понятно, что вызывающий код становится владельцем датасета и отвечает за его время жизни. Передача владения явно выражается через std::move.

Важно отметить, что использование std::unique_ptr не влечёт значительных накладных расходов по сравнению с сырыми указателями. В случае с пользовательским удалителем, который передаётся как часть типа, размер std::unique_ptr остаётся равным размеру сырого указателя, а вызов удалителя в деструкторе не требует дополнительных косвенных обращений. Это позволяет применять предложенный подход даже в высоконагруженных сценариях, где критичны производительность и экономия памяти [407].

Таким образом, RAII-обёртки не только устраняют проблему утечек ресурсов, но и делают код самодокументированным, а управление ресурсами — предсказуемым.

Предложенный подход легко расширить на другие GDAL-объекты: OGRSpatialReference (освобождается через OGRSpatialReference::DestroySpatialReference), OGRGeometry (освобождается через OGRGeometryFactory::destroyGeometry) и другие. Для каждого достаточно определить свой функтор-удалитель (discarding functor) и соответствующий алиас для std::unique_ptr и добавить фабричные функции при необходимости. Полный список реализованных обёрток представлен в проекте геоинформационной системы [407], рис. 1.

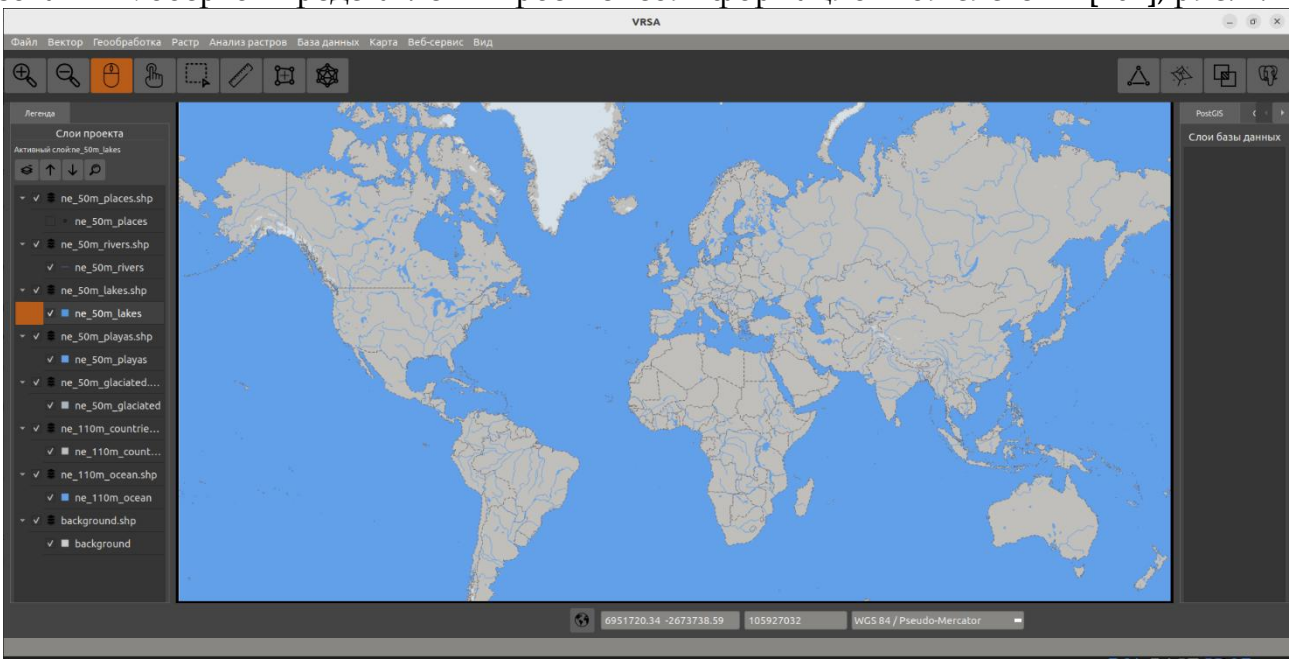

Рис. 1 – Визуализация пространственных данных Natural Earth в разработанном ГИС-приложении

Заключение

В данной статье были рассмотрены проблемы ручного управления ресурсами при использовании C++ API библиотеки GDAL, возникающие из-за C-стиля интерфейса. Показано, что такой подход приводит к усложнению кода, повышению риска утечек памяти и ненадёжности при обработке исключений.

В качестве решения предложено применение идиомы RAII и стандартных умных указателей std::unique_ptr с пользовательскими удалителями. Разработанные обёртки (GdalDatasetPtr, OgrFeaturePtr и другие) обеспечивают автоматическое освобождение ресурсов, делая код более безопасным, читаемым и самодокументированным.

Предложенный подход легко расширяется на другие GDAL-объекты.

Реализация обёрток доступна в открытом репозитории, что позволяет использовать их в собственных проектах.

Список литературы:

1. GDAL — Geospatial Data Abstraction Library [Электронный ресурс]. URL: https://gdal.org (дата обращения: 16.04.2026)
2. Software using GDAL — GDAL documentation [Электронный ресурс]. URL: https://gdal.org/software_using_gdal.html (дата обращения: 16.04.2026)
3. Vector API tutorial — GDAL documentation [Электронный ресурс]. URL: https://gdal.org/en/stable/tutorials/vector_api_tut.html (дата обращения: 16.04.2026)
4. Мейерс, Скотт. Эффективный и современный C++ [Текст]: 42 рекомендации по использованию C++11 и C++14.: Пер. С англ. — Спб. : ООО «Диалектика», 2023. — 304 с. : ил. — Парал. тит. англ (дата обращения 12.04.2026)
5. Лукин И.П. Реализация RAII-обёрток для GDAL в проекте VRSA [Электронный ресурс]. URL: https://github.com/Doggerr111/vrsa-gis/blob/main/gdal/gdalresourcehandles.h (дата обращения: 16.04.2026).

References:

1. GDAL — Geospatial Data Abstraction Library [Electronic resource]. URL: https://gdal.org (date of access: 16.04.2026)
2. Software using GDAL — GDAL documentation [Electronic resource]. URL: https://gdal.org/software_using_gdal.html (date of access: 16.04.2026)
3. Vector API tutorial — GDAL documentation [Electronic resource]. URL: https://gdal.org/en/stable/tutorials/vector_api_tut.html (accessed: 16.04.2026)
4. Meyers, Scott. Effective and Modern C++ [Text]: 42 Recommendations for Using C++11 and C++14.: Per. From English. — St. Petersburg: Dialectics LLC, 2023. — 304 p. : ill. — Paral. tit. Eng (accessed on 12.04.2026)
5. Lukin I.P. Implementation of RAII-wrappers for GDAL in the VRSA project [Electronic resource]. URL: https://github.com/Doggerr111/vrsa-gis/blob/main/gdal/gdalresourcehandles.h (accessed: 16.04.2026).