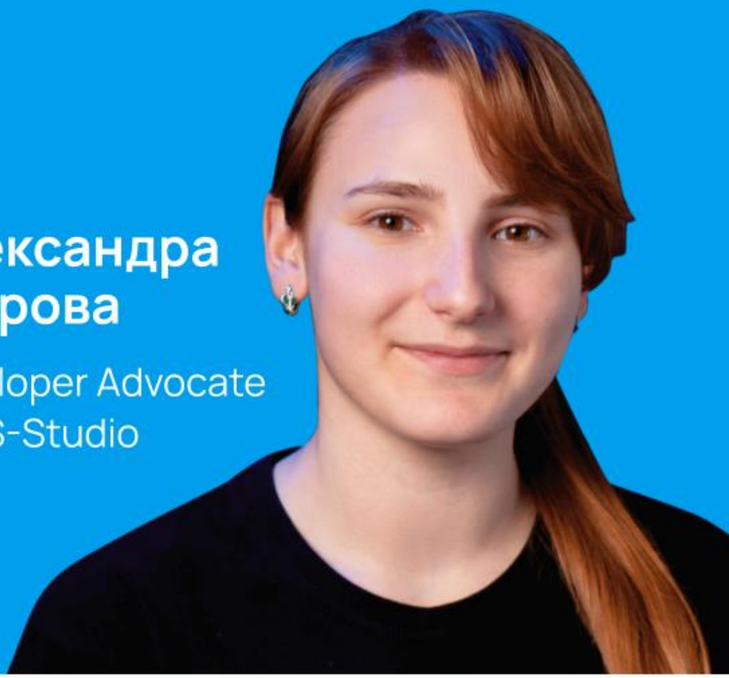
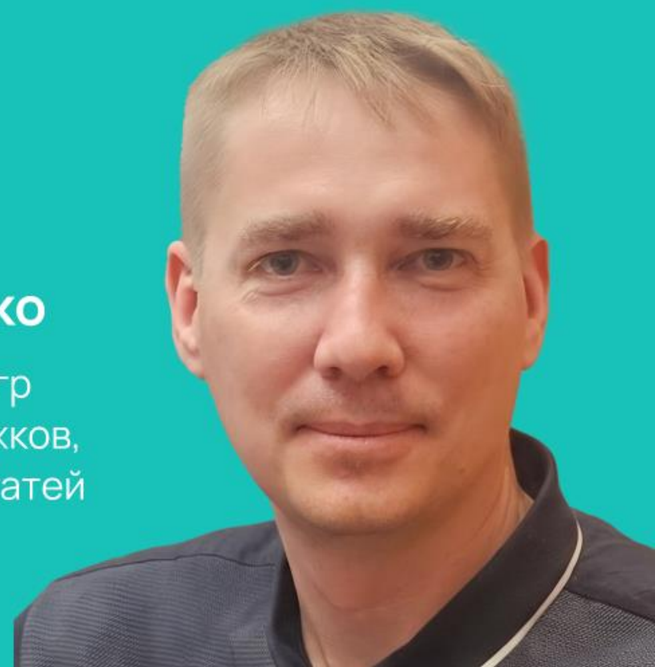
ВЕБИНАР

Каждая идиома когда-то была проблемой



Александра Уварова

Developer Advocate в PVS-Studio



Сергей Кушниренко

Разработчик игр и игровых движков, автор книг и статей

PVS-Studio

Спикеры и эксперты вебинара

Александра Уварова

Developer Advocate

- Разработчик C++ анализатора PVS-Studio
- Рассказываю про качество и безопасность кода на конференциях
- Пишу технические статьи о C++, статическом анализе и ошибках в реальном коде

PVS-Studio

Сергей Кушниренко

Разработчик игр и игровых движков, автор книг и статей

- Разработчик игровых движков более 20 лет
- Работал над Metro: Exodus, Deathloop, Age of Empires II, Stellaris, Cuisine Royale и The Sims Mobile / SimCity BuildIt
- Ведёт open-source проект Akhenaten по восстановлению Pharaoh (1999)
- Автор курса «C++ без аллокаций» на Stepik, а также книг и статей про игровые движки и высокопроизводительный код

https://dalerank.github.io/

Каждая идиома когда-то была проблемой

Формат

- Рассмотрим фрагменты кода реальных проектов
- Найдем в них проблемные места и способы исправления
- Рассмотрим более 10 идиом
- Обсудим их применение в GameDev
- Ответим на вопросы

Немного деталей

- **PVS-Studio** – статический анализатор для поиска потенциальных уязвимостей и ошибок в коде
- Работает с языками C, C++, C#, Java, Go, JS и TS
- В целях популяризации мы анализируем некоторые Open Source проекты, находим в них ошибки и пишем статьи

Ошибок: 17089
Примеров: 6717
Проектов: 576

База ошибок, найденных PVS-Studio,
в коде открытых проектов

https://pvs-studio.ru/ru/blog/examples/

Каждая идиома когда-то была проблемой

проблемой

DPDK
DATA PLANE DEVELOPMENT KIT

```
static int
cnxk_gpio_read_attr(char *attr, char *val)
{
  FILE *fp;
  int ret;

  fp = fopen(attr, "r");
  if (!fp)
    return -errno;

  ret = fscanf(fp, "%s", val);
  if (ret < 0)
    return -errno;
  // ....
}
```

V773 The function was exited without closing the file referenced by the 'fp' handle. A resource leak is possible.

Мы уже нашли 207 похожих ошибок


```
std::vector<not_null<PeerData*>> _list;

void RecentPeers::applyLocal(QByteArray serialized)
{
  _list.clear();
  // ....
  _list.reserve(count);
  for (auto i = 0; i != int(count); ++i)
  {
    // ....
    if (stream.ok() && peer) {
      _list.push_back(peer);
    } else {
      _list.clear();
      DEBUG_LOG(("Suggestions: Failed RecentPeers reading %1 / %2."
        ).arg(i + 1
        ).arg(count));
      _list.clear();
      return;
    }
  }
  DEBUG_LOG(
    ("Suggestions: RecentPeers read OK, count: %1").arg(_list.size()));
}
```

V586 The 'clear' function is called twice for deallocation of the same resource.

Мы уже нашли 8 похожих ошибок

RAII

Resource Acquisition Is Initialization

одна из фундаментальных идиом C++, которая делает управление ресурсами автоматическим и безопасным

RAII / почему название неудачное

Термин ввёл Бьёрн Страуструп на заре языка. По его же признанию, название получилось неудачным: оно подчёркивает лишь половину идеи (захват ресурса в конструкторе) и совсем не акцентирует вторую, куда более важную часть **освобождение в деструкторе.**

Более точные имена

- **Scoped Locking — для мьютексов (Дуглас Шмидт)**
- **Execute-Around Object — для обёрток, делающих что-то на входе и выходе из области видимости**

Главная ценность RAII не в том, что вы не забудете `free` или `unlock`, а что освобождение корректно работает **при исключениях.**

Без RAII любой `throw` между захватом и ручным освобождением — это утечка.

RAII / в играх

«открыл / закрыл»

ScopedTimer

Измеряет время куска кадра и пишет результат в профайлер прямо в деструкторе.

RenderPassScope

Открывает render pass в конструкторе и закрывает в деструкторе — никаких ручных пар вызовов.

CommandBuffer

Привязки состояния GPU и командные буферы освобождаются автоматически при выходе из области видимости.

StreamingLock

Блокировка на время доступа к стримингу ресурсов из фонового потока снимается гарантированно.

RAII / правильно

```
class ScopedTextureBind {
public:
    explicit ScopedTextureBind(GLuint texture) {
        glBindTexture(GL_TEXTURE_2D, texture); // захват
    }
    ~ScopedTextureBind() {
        glBindTexture(GL_TEXTURE_2D, 0); // освобождение — всегда
    }
    // Запрещаем копирование: ресурс должен быть один хозяин
    ScopedTextureBind(const ScopedTextureBind&) = delete;
    ScopedTextureBind& operator=(const ScopedTextureBind&) = delete;
};

void draw_hud() {
    ScopedTextureBind bind(hud_atlas);
    submit_quads();
    // даже если тут вылетит исключение —
    // текстура будет отвязана при выходе из области видимости
}
```

RAII / неправильно

```
void draw_hud() {
    glBindTexture(GL_TEXTURE_2D, hud_atlas);
    if (!hud_visible)
        return;                         // текстура осталась привязанной
    submit_quads();                     // а если отсюда вылетит исключение —
    glBindTexture(GL_TEXTURE_2D, 0);    // сюда мы просто не дойдём
}
```

Привязанная текстура останется активной при раннем выходе или утечёт при исключении.

```
~ScopedFile() {
    if (fclose(f_) != 0)
        throw IOError("не смог закрыть");   // билет в один конец
}                                           // до станции std::terminate
```

Бросать исключения из деструктора одна из самых частых ошибок. Это почти всегда приводит к немедленному завершению программы.

RAII / проблемы

RAII отлично гарантирует, что деструктор вызовется.

Он ничего не гарантирует про то, **сколько этот деструктор будет работать, и в хотпасе цена «незаметного» кода в деструкторе становится очень заметной.**

Scope guard

Scope Guard

Scope Guard - это RAII, доведённый до логического завершения. Вместо отдельного класса-обёртки под каждый ресурс, Scope Guard представляет собой универсальный объект, которому в конструктор передают **произвольное действие (лямбду, функтор, указатель на функцию), выполняемое в деструкторе.**

Таким образом, получается «отложенный код», гарантированно срабатывающий при выходе из области видимости, что идеально подходит для обеспечения корректной очистки ресурсов.

Scope guard / История

Идиома прошла путь от популяризации до стандартизации, адаптируясь под новые парадигмы языка.

Ранние 2000-е

Популяризация Андреем Александреску и Петру Маргининым, введение самого названия.

Move-семантика

Александреску переосмысливает RAII, добавляя `SCOPE_EXIT` / `SCOPE_FAIL` / `SCOPE_SUCCESS`.

Современность

Использование в библиотеке Folly от Facebook и включение в `std::experimental::scope_exit`.

Scope Guard / где живёт в геймдеве

Многошаговая загрузка

При сбое декомпрессии требуется откатить изменения: вернуть память в пул и снять зарегистрированные хендлы, иначе пул течёт, а менеджер ресурсов ссылается на мусор.

Тулчейны и редакторы уровней

Операции с откатом критичны: импорт ассета должен примениться целиком или не оставить следов, чтобы поддерживать целостность данных.

Транзакционная логика пишется **линейно, сверху вниз, без вложенных «лестниц очистки», что резко снижает количество багов.**

Scope Guard / правильно

```
template <class F>
class ScopeGuard {
    F func_;
    bool active_ = true;
public:
    explicit ScopeGuard(F f) : func_(std::move(f)) {}
    ~ScopeGuard() { if (active_) func_(); }

    void dismiss() { active_ = false; }

    ScopeGuard(ScopeGuard&& o) : func_(std::move(o.func_)), active_(o.active_) {
      o.dismiss();
    }
};
```

Scope Guard / неправильно

```
void load_level(Level& lvl) {
    auto* mem = pool.allocate(lvl.size);

    make_guard([&]{ pool.free(mem); });   // guard без имени
    // это временный объект — он умирает в конце ЭТОЙ ЖЕ строки,
    // память уже освобождена, дальше работаем с висячим указателем

    decompress_into(mem, lvl.archive);    // пишем в чужую память
}
```

```
void import_asset(const Path& p) {
    auto rollback = make_guard([&]{ db.remove(p); });
    db.insert(p);

    bake_textures(p); // забыли rollback.dismiss() успешный импорт откатывается сам
}
```

Smart pointer

Smart pointer / откуда взялась

Прямое применение идиомы RAII к самому распространённому ресурсу (памяти из кучи). Smart pointer ведет себя как обычный указатель (`*`, `->`), но при этом автоматически управляет временем жизни того, на что указывает, гарантируя корректное освобождение ресурсов.

1

Конец 90-х

`std::auto_ptr` с «копированием-перемещением» оказался неудачным и был удален из стандарта.

2

Начало 2000-х

Библиотека Boost переосмыслила идею, представив `scoped_ptr`, `shared_ptr` и `weak_ptr`.

3

C++11

Отшлифованные годами применения реализации Boost вошли в стандарт, став неотъемлемой частью современного C++.

Деление по политике владения

`unique_ptr`

Единоличный владелец, только семантика перемещения (move), не несет накладных расходов во время выполнения.

`shared_ptr`

Разделяемое владение через счётчик ссылок, объект удаляется, когда последний `shared_ptr` перестает на него ссылаться.

`weak_ptr`

Наблюдатель, который позволяет безопасно получить доступ к объекту, но не продлевает его жизнь. Используется для разрыва циклических ссылок между `shared_ptr`.

Smart pointer / в играх

`unique_ptr`

Используется свободно и широко. Не имеет накладных расходов во время выполнения, кроме стоимости деструктора, и четко выражает **единоличное владение ресурсом.**

`shared_ptr`

Многие движки ограничивают или запрещают. Атомарный счетчик ссылок добавляет расходов, а разделяемое владение **размывает ответственность за время жизни объекта, что ведет к трудноуловимым утечкам.**

Smart pointer / правильно

```
class SoundBank {
    std::unordered_map<std::string, std::shared_ptr<AudioClip>> clips_;
public:
    std::shared_ptr<AudioClip> get(const std::string& name) {
        auto& slot = clips_[name];
        if (!slot)
            slot = std::make_shared<AudioClip>(load_from_disk(name));
        return slot;   // вызывающий разделяет владение с банком
    }
};

// unique_ptr и единоличное владение, ноль накладных расходов
std::unique_ptr<ParticleSystem> ps = std::make_unique<ParticleSystem>(1024);

// в хотпасе по ссылке, без копии и без атомарного инкремента
void mix(const std::shared_ptr<AudioClip>& clip);
```

Smart pointer / неправильно

```
// 1) копия shared_ptr на каждый вызов — атомарный инкремент в хотпасе
void mix(std::shared_ptr<AudioClip> clip);      // по значению
for (auto& voice : voices_) mix(voice.clip);    // много раз за кадр

// 2) цикл: родитель держит детей, ребёнок держит родителя
struct Node {
    std::shared_ptr<Node>              parent;   // <- вот тут нужен weak_ptr
    std::vector<std::shared_ptr<Node>> children;
};
// счётчики никогда не дойдут до нуля, поддерево не умрёт никогда

// 3) вышли из-под защиты и потеряли владельца
AudioClip* raw = bank.get("shot").get();   // временный shared_ptr умер
raw->play();                               // висячий указатель
```

```
class MaybeUtf8
{
  // ....
  private:
    void AllocateSufficientSpace(int len)
    {
      if (len + 1 > MAX_STACK_LENGTH)
      {
        allocated_.reset(new uint8_t[len + 1]);
        buf_ = allocated_.get();
      }
    }
    // ....
    std::unique_ptr<uint8_t> allocated_;
}
```

V554 Incorrect use of unique_ptr. The memory allocated with 'new []' will be cleaned using 'delete'.

Мы уже нашли 24 похожие ошибки

```
class MaybeUtf8
{
  // ....
  private:
    void AllocateSufficientSpace(int len)
    {
      if (len + 1 > MAX_STACK_LENGTH)
      {
        allocated_.reset(new uint8_t[len + 1]);
        buf_ = allocated_.get();
      }
    }
    // ....
    std::unique_ptr<uint8_t[]> allocated_;
}
```

V554 Incorrect use of unique_ptr. The memory allocated with 'new []' will be cleaned using 'delete'.

Мы уже нашли 24 похожие ошибки

Checked Delete / откуда взялась

Проблема возникает, когда вы пытаетесь удалить указатель (`delete p`) на тип, который был объявлен только через предварительное объявление (forward declaration), но его полное определение отсутствует в этой точке кода.

Компилятор, не видя полного определения класса, не знает, есть ли у него деструктор. В результате он просто освобождает память, но **не вызывает деструктор объекта. В лучшем случае вы получите предупреждение компилятора, которое легко пропустить, а в худшем это приведет к незаметной утечке всех ресурсов, которыми владел объект (например, файловых дескрипторов, сетевых соединений или другой динамической памяти).**

Checked Delete / в играх

Границы модулей и в Pimpl-обёртках, Время сборки большого движка,это отдельная боль, измеряемая часами.

Руками эту идиому почти никто не пишет и стандартные умные указатели делают это за вас. Но без неё невозможно объяснить, почему `unique_ptr` на Pimpl-типе **требует, чтобы деструктор класса был объявлен в заголовке и определён в `.cpp`, где тип полный.**

Checked Delete / правильно

```
template <class T>
inline void checked_delete(T* p) {
    // если T неполный, sizeof не скомпилируется
    using complete = char[sizeof(T) ? 1 : -1];
    (void) sizeof(complete);
    delete p;
}
```

```
// physics_world.h
class PhysicsWorld {
    struct Impl;                        // неполный тип (forward declaration)
    std::unique_ptr<Impl> impl_;


public:
    PhysicsWorld();
    ~PhysicsWorld();                    // ОБЪЯВЛЕН здесь определён в .cpp
};


// physics_world.cpp
struct PhysicsWorld::Impl { /* всё настоящее наполнение */ };
    PhysicsWorld::~PhysicsWorld() = default;   // тут Impl полный, всё честно
}
```

Checked Delete / неправильно

```
// physics_world.h
class PhysicsWorld {
    struct Impl;
    std::unique_ptr<Impl> impl_;

public:
    PhysicsWorld();
    // деструктора нет компилятор сгенерирует его ЗДЕСЬ,
    // где Impl ещё неполный
};
```


```
TransformIterator& operator--(int) {
  TransformIterator tmp(*this);
  --*this;
  return tmp;
}

TransformIterator& operator++(int) {
  TransformIterator tmp(*this);
  ++*this;
  return tmp;
}
```

V558 Function returns the reference to temporary local object: tmp.

Мы уже нашли 11 похожих ошибок


```
TransformIterator operator--(int) {
  TransformIterator tmp(*this);
  --*this;
  return tmp;
}

TransformIterator operator++(int) {
  TransformIterator tmp(*this);
  ++*this;
  return tmp;
}
```

V558 Function returns the reference to temporary local object: tmp.

Мы уже нашли 11 похожих ошибок


```
template<typename T> struct Ptr : public std::shared_ptr<T>;
// ....
Ptr<FlannNeighborhoodGraph> FlannNeighborhoodGraph::create(....)
{
    return makePtr<FlannNeighborhoodGraphImpl>(....);
}
```

V758 The 'graph' reference becomes invalid when smart pointer returned by a function is destroyed.

```
template<typename T> struct Ptr : public std::shared_ptr<T>;
// ....
Ptr<FlannNeighborhoodGraph> FlannNeighborhoodGraph::create(....)
{
    return makePtr<FlannNeighborhoodGraphImpl>(....);
}

void Utils::densitySort
  (/* .... */)
{
  // ....
  FlannNeighborhoodGraph &graph = *FlannNeighborhoodGraph::create(....);   // <=

  std::vector<double> sum_knn_distances (points_size, 0);
  for (int p = 0; p < points_size; p++) {
    const std::vector<double> &dists = graph.getNeighborsDistances(p);
    for (int k = 0; k < knn; k++)
      sum_knn_distances[p] += dists[k];
  }
  // ....
}
```

V758 The 'graph' reference becomes invalid when smart pointer returned by a function is destroyed.

```
template<typename T> struct Ptr : public std::shared_ptr<T>;
// ....
Ptr<FlannNeighborhoodGraph> FlannNeighborhoodGraph::create(....)
{
    return makePtr<FlannNeighborhoodGraphImpl>(....);
}

void Utils::densitySort
  (/* .... */)
{
  // ....
  FlannNeighborhoodGraph &graph = *FlannNeighborhoodGraph::create(....);

  std::vector<double> sum_knn_distances (points_size, 0);
  for (int p = 0; p < points_size; p++) {
    const std::vector<double> &dists = graph.getNeighborsDistances(p);   // <=
    for (int k = 0; k < knn; k++)
      sum_knn_distances[p] += dists[k];
  }
  // ....
}
```

V758 The 'graph' reference becomes invalid when smart pointer returned by a function is destroyed.

COMPILER EXPLORER Add... More Templates Sponsors intel Solid Sands JETBRAINS Share Policies

C++ source #1

```
1  #include <iostream>
2  #include <memory>
3
4  class A
5  {
6  public:
7      int a = 10;
8  };
9
10 std::shared_ptr<A> Foo()
11 {
12   std::shared_ptr<A> pa(new A());
13   return pa;
14 }
15  int main()
16 {
17   const A &ra = *Foo();
18   std::cout << ra.a;
19 }
```

PVS-Studio x86-64 clang (trunk) (Editor #1, Compiler #1)

Wrap lines | Arguments | Stdin

The documentation for all analyzer warnings is available here:
https://pvs-studio.com/en/docs/warnings/.

<source>:17:1: error: V758 The 'ra' reference becomes invalid when smart pointer returned by a function is destroyed.

Output of x86-64 clang (trunk) (Compiler #1) | x86-64 clang (trunk) (Editor #1)

Wrap lines | Select all

x86-64 clang (trunk) -O3 -fsanitize=address

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 1
=================================================================
==1==ERROR: AddressSanitizer: heap-use-after-free on address 0x6d1d111e0010 at pc 0x62aa7d82f133 bp 0x7ffcf6927720 sp 0x7ffcf6927718
READ of size 4 at 0x6d1d111e0010 thread T0
    #0 0x62aa7d82f132 in main /app/example.cpp:18:19
    #1 0x70fd11e2a1c9  (/lib/x86_64-linux-gnu/libc.so.6+0x2a1c9) (BuildId: 328820b908de8ea1ef79afa8995e302e819163d7)
    #2 0x70fd11e2a28a in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x2a28a) (BuildId: 328820b908de8ea1ef79afa8995e302e819163d7)
    #3 0x62aa7d73d3b4 in _start (/app/output.s+0x2e3b4)

0x6d1d111e0010 is located 0 bytes inside of 4-byte region [0x6d1d111e0010,0x6d1d111e0014)
freed by thread T0 here:
    #0 0x62aa7d82e642 in operator delete(void*, unsigned long) /root/llvm-project/compiler-rt/lib/asan/asan_new_delete.cpp:187:3
    #1 0x62aa7d82f05e in std::_Sp_counted_base<(__gnu_cxx::_Lock_policy)2>::_M_release() /opt/compiler-explorer/gcc-snapshot/lib/gcc/x86_64-linux-
gnu/17.0.0/../../../../include/c++/17.0.0/bits/shared_ptr_base.h:423:8
    #2 0x62aa7d82f05e in std::__shared_count<(__gnu_cxx::_Lock_policy)2>::~__shared_count() /opt/compiler-explorer/gcc-snapshot/lib/gcc/x86_64-linux-
gnu/17.0.0/../../../../include/c++/17.0.0/bits/shared_ptr_base.h:1101:11
    #3 0x62aa7d82f05e in std::__shared_ptr<A, (__gnu_cxx::_Lock_policy)2>::~__shared_ptr() /opt/compiler-explorer/gcc-snapshot/lib/gcc/x86_64-linux-
gnu/17.0.0/../../../../include/c++/17.0.0/bits/shared_ptr_base.h:1575:25
    #4 0x62aa7d82f05e in main /app/example.cpp:17:17
    #5 0x70fd11e2a28a in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x2a28a) (BuildId: 328820b908de8ea1ef79afa8995e302e819163d7)
    #6 0x62aa7d73d3b4 in _start (/app/output.s+0x2e3b4)

previously allocated by thread T0 here:
    #0 0x62aa7d82d9ed in operator new(unsigned long) /root/llvm-project/compiler-rt/lib/asan/asan_new_delete.cpp:109:35
    #1 0x62aa7d82edf4 in Foo() /app/example.cpp:12:25
    #2 0x70fd11e2a28a in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x2a28a) (BuildId: 328820b908de8ea1ef79afa8995e302e819163d7)
    #3 0x62aa7d73d3b4 in _start (/app/output.s+0x2e3b4)

SUMMARY: AddressSanitizer: heap-use-after-free /app/example.cpp:18:19 in main
Shadow bytes around the buggy address:
  0x6d1d111dfd80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x6d1d111dfe00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x6d1d111dfe80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x6d1d111dff00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x6d1d111dff80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x6d1d111e0000: fa fa[fd]fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x6d1d111e0080: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
```

OpenCV

```
void Utils::densitySort
  (const Mat &points, int knn, Mat &sorted_points, std::vector<int> &sorted_mask)
{
  // ....
  FlannNeighborhoodGraph &graph = *FlannNeighborhoodGraph::create(....);

  std::vector<double> sum_knn_distances (points_size, 0);
  for (int p = 0; p < points_size; p++) {
    const std::vector<double> &dists = graph.getNeighborsDistances(p);
    for (int k = 0; k < knn; k++)
      sum_knn_distances[p] += dists[k];
  }
  // ....
}
```

V758 The 'graph' reference becomes invalid when smart pointer returned by a function is destroyed.

```
void Utils::densitySort
  (const Mat &points, int knn, Mat &sorted_points, std::vector<int> &sorted_mask)
{
  // ....
  sstd::vector<double> sum_knn_distances (points_size, 0);
  {
    auto graph = FlannNeighborhoodGraph::create(....);
    for (int p = 0; p < points_size; p++) {
      const std::vector<double> &dists = graph->getNeighborsDistances(p);
      for (int k = 0; k < knn; k++)
        sum_knn_distances[p] += dists[k];
    }
  }
  // ....
}
```

V758 The 'graph' reference becomes invalid when smart pointer returned by a function is destroyed.

Resource return

Resource Return / откуда взялась

Проблема владения ресурсами

Идиома возникла для обеспечения безопасной передачи **владения ресурсом от функции вызывающему коду.**

В традиционном подходе возврат сырого указателя часто приводил к проблемам вроде утечкек памяти, двойным удалениям или потере указателя при исключениях.

`auto_ptr`

В 90-х `std::auto_ptr` пытался решить эту задачу. Однако его "копирование как перемещение" создавало множество неочевидных ловушек, что привело к его признанию ошибкой дизайна и последующему удалению из стандарта C++.

C++11: move

С появлением move-семантики в C++11 идиома "Resource Return" наконец-то обрела своё полноценное и безопасное воплощение.

Move-семантика позволила правильно реализовать передачу владения, и теперь современный программист просто использует умные указатели, не задумываясь о скрытых сложностях.

Resource Return / в играх

Владеющие хендлы

Меши, текстуры, звуки и материалы в игровых движках используются через **владеющие хендлы или умные указатели. Это гарантирует, что ресурс будет корректно управляться с момента его создания.**

Безопасная загрузка

Если процесс загрузки прерывается или завершается ошибкой, частично загруженный ресурс **корректно освободится сам, предотвращая утечки и некорректное состояние.**

Лёгкие хендлы-значения

Многие движки используют **лёгкие хендлы-значения (например, `TextureHandle` с индексом и поколением).**

Это дополнительно упрощает передачу и управление, сохраняя при этом семантику владения.

Resource Return / правильно

```
// возвращает владение явно и безопасно
std::unique_ptr<Mesh> load_mesh(const std::string& path) {
    auto mesh = std::make_unique<Mesh>();
    mesh->upload(read_vertices(path));
    return mesh;            // move, ни одной лишней копии буфера вершин
}

void build_scene(Scene& scene) {
    scene.add(load_mesh("rock.obj"));   // владение перетекает в сцену
    // потерять меш физически негде
    // либо он в scene, либо в temp, который мувнулся
}
```

Resource Return / неправильно

```
// caller must delete!  // <- только документация вместо контракта
Mesh* load_mesh(const std::string& path) {
    Mesh* mesh = new Mesh();
    mesh->upload(read_vertices(path));   // а если тут исключение
    return mesh;                         // mesh утёк, и никто не узнает
}

void build_scene(Scene& scene) {
    Mesh* m = load_mesh("rock.obj");
    if (!scene.has_room())
        return;                          // утечка на раннем выходе

    scene.add(m);                        // а кто теперь владелец? scene? мы?
}
```

```
struct EinsumParams {
  int inputSize;
  int outputSize;
  std::string equation;
  std::vector<MatShape> einsumInpShapes;
  EinsumParams(std::string equation_,
    std::vector<MatShape> einsumInpShapes_ = std::vector<MatShape>())
  {
    inputSize = einsumInpShapes_.size();
    equation = equation_;
    einsumInpShapes = einsumInpShapes_;
  }
};
```

V820 The 'equation_' variable is not used after copying. Copying can be replaced with move/swap for optimization.

```
struct EinsumParams {
  int inputSize;
  int outputSize;
  std::string equation;
  std::vector<MatShape> einsumInpShapes;
  EinsumParams(std::string equation_,
    std::vector<MatShape> einsumInpShapes_ = std::vector<MatShape>())
  {
    inputSize = einsumInpShapes_.size();
    equation = std::move(equation_);
    einsumInpShapes = std::move(einsumInpShapes_);
  }
};
```

V820 The 'equation_' variable is not used after copying. Copying can be replaced with move/swap for optimization.

```
struct EinsumParams {
  int inputSize;
  int outputSize;
  std::string equation;
  std::vector<MatShape> einsumInpShapes;
  EinsumParams(std::string equation_,
    std::vector<MatShape> einsumInpShapes_ = std::vector<MatShape>())
    : inputSize(einsumInpShapes_.size())
    , equation(std::move(equation_))
    , einsumInpShapes(std::move(einsumInpShapes_))
  {
  }
};
```

V820 The 'equation_' variable is not used after copying. Copying can be replaced with move/swap for optimization.

Move ctor

Move ctor / откуда взялась

До C++11: Дорогие копии

Возврат тяжёлого объекта из функции или вставка его в контейнер означали **реальную глубокую копию.** Попытки решить это приводили к ухищрениям вроде проблемного `auto_ptr`.

Суть move-семантики

В C++11 появилось понятие «исходник, у которого можно отобрать содержимое». Rvalue-ссылки (`T&&`) отличают временные объекты, а move-конструктор забирает их ресурсы, обнуляя источник.

Что делает `std::move`

`std::move` **ничего не перемещает.** Это всего лишь приведение к rvalue-ссылке, которое сообщает компилятору: «считай этот объект отдаваемым, его содержимое можно безопасно перенести».

Значимость и влияние

Потенциально дорогая глубокая копия превращается в дешёвый **обмен нескольких указателей.** Move-семантика, разработанная Говардом Хиннантом, Бьёрном Страуструпом и Дэйвом Абрахамсом, стала фундаментом современной стандартной библиотеки C++.

Move ctor / в играх

Игровой код очень хорошо принял move-семантику, поскольку он постоянно перемещает тяжёлые объекты (меши, текстуры, звуковые сэмплы, контейнеры с данными кадра).

НО! В большинстве движков возврат таких объектов из загрузчика или их вставка в контейнер уже осуществлялись по-своему. Через хендлы, пулы или ручную передачу владения и многие разработчики не стали отказываться от этих механизмов, а просто добавили move-семантику как обёртку над ними.

Три вещи, за которые платим:

1 «Пустое, но валидное» состояние источника

После перемещения исходный объект находится в состоянии, когда из него нельзя читать осмысленные данные, но его деструктор и операции присваивания обязаны корректно работать.

2 `noexcept` обязателен

Для того чтобы контейнеры стандартной библиотеки могли эффективно использовать перемещение, move-конструктор должен быть помечен как `noexcept`. В противном случае они откатываются к более дорогостоящему копированию.

3 Тонкие правила автогенерации

Объявление собственного деструктора или любой из копирующих операций **подавляет автоматическую генерацию move-конструктора и move-оператора присваивания, и «перемещение» превращается в копирование.**

Move Constructor / правильно

```
class Mesh {
    std::size_t count_ = 0;
    Vertex* verts_ = nullptr;

public:
    // Move-конструктор: крадём буфер источник обнуляем. noexcept обязателен
    Mesh(Mesh&& o) noexcept : count_(o.count_), verts_(o.verts_) {
        o.count_ = 0; o.verts_ = nullptr;
        // источник пустой, но валидный
    }
    Mesh& operator=(Mesh&& o) noexcept {
        delete[] verts_; // Освобождаем старые ресурсы текущего объекта
        count_ = o.count_; verts_ = o.verts_;
        o.count_ = 0; o.verts_ = nullptr;
        return *this;
    }
    ~Mesh() { delete[] verts_; }
};

std::vector<Mesh> meshes;
meshes.push_back(load());   // меш перемещается в вектор без копии буфера вершин
```

Move Constructor / неправильно

```
// 1) move без noexcept — vector при реаллокации откатится на КОПИРОВАНИЕ
Mesh(Mesh&& o) : count_(o.count_), verts_(o.verts_) { ... }
//            ^ здесь молча теряется вся оптимизация

// 2) забыли обнулить источник двойное освобождение
Mesh(Mesh&& o) noexcept : count_(o.count_), verts_(o.verts_) {}
// у обоих объектов один verts_, оба вызовут delete[]

// 3) объявили деструктор и move не сгенерируется, будет копирование
class Chunk {
    std::vector<Vertex> data_;
    ~Chunk() { log("chunk died"); }    // одна строчка логирования
};                                     // и весь класс перестал перемещаться

// 4) читаем после перемещения
auto moved = std::move(mesh);
mesh.draw();                           // объект валиден, но пуст
```

`move` ничего не перемещает

`const` не гарантирует константность

`static` означает пять разных вещей в зависимости от места

C++ язык, где имена выбирали по остаточному принципу.

```
class SQLQueryPrivate
{
public:
  std::weak_ptr<SQLDatabase> _db_ref;
  sqlite3 * _db;
  sqlite3_stmt * _stmt;
  bool _hasNext;
  std::string _query;

  SQLQueryPrivate(std::shared_ptr<SQLDatabase> db_ref,
                  sqlite3 * db, sqlite3_stmt * stmt,
                  const std::string& query);
  SQLQueryPrivate(const SQLQueryPrivate& other);
  ~SQLQueryPrivate();
};

SQLQueryPrivate::SQLQueryPrivate(const SQLQueryPrivate& other) :
    _db_ref(other._db_ref),
    _db(other._db),
    _stmt(other._stmt),
    _hasNext(other._hasNext),
    _query(other._query)
{
  const_cast<SQLQueryPrivate &>(other)._stmt = NULL;
}
```

V690 The 'SQLQueryPrivate' class implements a copy constructor, but lacks the '=' operator. It is dangerous to use such a class.

Мы уже нашли 17 похожих ошибок

```
IUnversionedRowBatchPtr
THorizontalSchemalessRangeChunkReader::Read(
    const TRowBatchReadOptions& options)
{
    TCurrentTraceContextGuard traceGuard(TraceContext_);
    auto readGuard = AcquireReadGuard();                    // <=
    // ....
}
```

V1002 The 'TTimerGuard' class, containing pointers, constructor and destructor, is copied by the automatically generated copy constructor.

```
template <class TTimer>
class TTimerGuard
{
public:
  explicit TTimerGuard(TTimer* timer)
  {
    Timer_->Start();
  }

  ~TTimerGuard()
  {
    Timer_->Stop();
  }

private:
  TTimer* const Timer_;
};
```

V1002 The 'TTimerGuard' class, containing pointers, constructor and destructor, is copied by the automatically generated copy constructor.

```cpp
template <class TTimer>
class TTimerGuard
{
public:
  explicit TTimerGuard(TTimer* timer)
  {
    Timer_->Start();
  }

  ~TTimerGuard()
  {
    Timer_->Stop();
  }

 TTimerGuard(const TTimerGuard &) = delete;
 TTimerGuard& operator=(const TTimerGuard &) = delete;

private:
  TTimer* const Timer_;
};
```

V1002 The 'TTimerGuard' class, containing pointers, constructor and destructor, is copied by the automatically generated copy constructor.

0 / 3 / 5

0 / 3 / 5 / откуда взялась

Правило трёх (эпоха C++98)

Если определён деструктор, конструктор копирования или оператор копирующего присваивания, то почти наверняка нужны все три.

Правило пяти (эпоха C++11)

К тройке добавились move-конструктор и move-присваивание. Иначе перемещение либо не сгенерируется и **станет копированием, либо сгенерируется некорректно.**

Правило нуля

Лучший способ соблюсти правила трёх и пяти **не писать ни одной из этих функций вообще. Современный C++ поощряет передачу владения ресурсами умным указателям и RAII-объектам.**

Авторы и популяризаторы

Правило трёх - Марк Клайн

Правило пяти оформилось с move-семантикой

Правило нуля - Р. Мартиньо Фернандес **в 2012 году.**

0 / 3 / 5 / в играх

Правило нуля

Если каждый член класса сам корректно управляет своим жизненным циклом, то компиляторные версии всех пяти специальных функций автоматически окажутся правильными.

Правило трёх/пяти

- GPU-буферы и текстуры (`GLuint`, `ID3D12Resource*`)
- Дескрипторы платформенных API (например, Vulkan)
- Ручные аллокации из пулов и арен памяти
- Всё, что пришло из C-библиотек

Правило деструктора

Даже пустой деструктор (даже `= default`) **подавляет автогенерацию move-конструктора и move-оператора присваивания у всего класса.**

0 / 3 / 5 / как правильно

```
// Правило пяти: класс с сырым ресурсом обязан определить все пять
class Buffer {
    float* data_; std::size_t size_;
public:
    // Деструктор: освобождает выделенную память
    ~Buffer() { delete[] data_; }                          // 1
    // Конструктор копирования: выполняет глубокое копирование
    Buffer(const Buffer&);                                 // 2
    // Оператор копирующего присваивания: обрабатывает самоприсваивание и выделение/освобождение
    Buffer& operator=(const Buffer&);                      // 3
    // Конструктор перемещения: забирает ресурсы у временного объекта
    Buffer(Buffer&&) noexcept;                             // 4
    // Оператор перемещающего присваивания: забирает ресурсы и освобождает старые
    Buffer& operator=(Buffer&&) noexcept;                  // 5
};
```

0 / 3 / 5 / как правильно

```
// Правило нуля: ничего не пишем и члены сами управляют собой, и это правильно
class Mesh {
    std::vector vertices_;
    // std::vector сам копируется, перемещается, чистится
    std::string name_;
    // std::string сам копируется, перемещается, чистится
    // Нет необходимости объявлять деструктор
    // copy- или move-конструкторы/операторы — компилятор сделает всё корректно
};
```

0 / 3 / 5 / как неправильно

```
class Buffer {
    float* data_;
    std::size_t size_;
public:
    Buffer(std::size_t n) : data_(new float[n]), size_(n) {}
    ~Buffer() { delete[] data_; }

    Buffer(const Buffer& o)                    // копирующий конструктор есть
        : data_(new float[o.size_]), size_(o.size_) {
        std::copy_n(o.data_, size_, data_);
    }
    // ...a operator= нет. Компилятор сгенерирует поверхностный.
};

Buffer a(1024), b(512);
a = b;          // скопировался указатель: утёк a.data_,
                // теперь оба владеют b.data_ и оба его удалят
```

> Правило нуля — единственное правило C++, которое выполняется бездействием.
>
> Наконец-то язык вознаграждает лень.

Non-copyable Mixin

Non-copyable Mixin / откуда взялась

Что это?

Маленький базовый класс, единственная задача которого — **запретить копирование того, кто от него наследуется.**

Это достигается за счёт удаления (или, до C++11, приватного и неопределенного) конструктора копирования и оператора присваивания.

Любой наследник автоматически теряет способность копироваться, компилятор не может сгенерировать копирующие операции производного класса

Зачем?

Объект, владеющий уникальным ресурсом (мьютекс, сокет, GPU-хендл, файл) при копировании породил бы **двух владельцев одного ресурса.**

Запрет копирования на уровне типа превращает потенциальную **ошибку времени выполнения в ошибку компиляции, что всегда дешевле.**

История и эволюция

До C++11 каноном был `boost::noncopyable`.

Скотт Майерс в «Effective C++» описывал эту идиому в приватно-необъявленном виде как способ «запретить функции, которые компилятор генерирует сам».

C++11 решил эту проблему с помощью ключевого слова `= delete`.

Non-copyable Mixin / в играх

Давнишний приём для всего, что владеет ресурсами или обязано существовать в единственном экземпляре:

Уникальные ресурсы

GPU-буферы, текстуры, командные очереди, файловые потоки, сетевые сессии, менеджеры подсистем. Случайное копирование таких объектов это **почти всегда баг.**

Предотвращение ошибок

Пометка non-copyable ловит эти баги на этапе компиляции, а не в виде загадочного двойного `glDeleteBuffers` в рантайме. Многие движки используют базовый класс-маркер (например, `FNoncopyable`).

Скрытая ловушка

Объявление копирующих операций, **даже удалённых (`= delete`), подавляет автогенерацию move-операций.**

Non-copyable тип по умолчанию становится ещё и **non-movable.**

Non-copyable Mixin / правильно

```
struct NonCopyable {
    NonCopyable() = default;
    NonCopyable(const NonCopyable&) = delete;             // Запрет копирующего конструктора
    NonCopyable& operator=(const NonCopyable&) = delete; // Запрет копирующего присваивания
};

class GpuBuffer : private NonCopyable {   // GpuBuffer нельзя скопировать
    GLuint id_;
    // Если необходима поддержка перемещения, её нужно явно объявить
public:
    GpuBuffer(GpuBuffer&&) noexcept;                      // Явный move-конструктор
    GpuBuffer& operator=(GpuBuffer&&) noexcept;          // Явный move-оператор присваивания
};
```

Non-copyable Mixin / неправильно

```
// 1) Запретили копирование и забыли про move
class GpuBuffer : private NonCopyable {
    GLuint id_;
};                                  // move тоже подавлен!
std::vector<GpuBuffer> buffers;
buffers.push_back(GpuBuffer{});     // Ошибка компиляции: тип не перемещается

// 2) Публичное наследование от класса-маркера
class GpuBuffer : public NonCopyable { /* ... */ };
// NonCopyable попал в публичный интерфейс, возможны неявные приведения
// к базе, у которой нет виртуального деструктора (что опасно)

// 3) До C++11: приватный, но необъявленный
class OldGpuBuffer {
    OldGpuBuffer(const OldGpuBuffer&);    // Объявлен, не определён
};
// Член класса или friend вызовет его и ошибка придёт только от ЛИНКОВЩИКА,
// текстом вида "unresolved external symbol", без единого намёка на причину
```

> `= delete` это самая продуктивная строчка в C++.
>
> Она ничего не делает, и именно поэтому работает.

```
namespace std
{
  inline void swap(cmList& lhs, cmList& rhs) noexcept
  {
    lhs.swap(rhs);
  }
} // namespace std
```

V1061 Extending the 'std' namespace may result in undefined behavior.

Мы уже нашли 10 похожих ошибок

```
class cmList
{
public:
  void swap(cmList& other) noexcept { /* implementation */ }
 // ....
};

inline void swap(cmList& lhs, cmList& rhs) noexcept
{
  lhs.swap(rhs);
}
```

V1061 Extending the 'std' namespace may result in undefined behavior.

Мы уже нашли 10 похожих ошибок

```
class cmList
{
public:
  void swap(cmList& other) noexcept { /* implementation */ }

  friend void swap(cmList& lhs, cmList& rhs) noexcept
  {
    lhs.swap(rhs);
  }
  // ....
};
```

V1061 Extending the 'std' namespace may result in undefined behavior.

Мы уже нашли 10 похожих ошибок

```cpp
template <typename T>
void foo(T &obj1, T &obj2)
{
  using std::swap;
  // ....
  swap(obj1, obj2);
  // ....
}
```

Трюк Barton-Nackman'a

Barton-Nackman / откуда взялась

Что это?

Метод определения свободных функций, например, `operator==`, непосредственно внутри шаблонного класса через `friend`-функцию.

Как это работает?

Дружественная функция автоматически создаётся для **каждой инстанциации шаблона и находится исключительно через ADL. Это предотвращает засорение глобального пространства имён и позволяет избежать конфликтов при разрешении перегрузок.**

Изначальная проблема

Ранние компиляторы C++ испытывали трудности с корректной работой и перегрузкой шаблонных функций. Трюк Barton-Nackman'a позволял обойти это, генерируя **нешаблонный оператор для каждой конкретной специализации типа.**

Происхождение имени

Идиома получила своё название по статье **Джона Бартона из начала 90-х годов**

Barton-Nackman / правильно

```
template <class T>
struct Comparable {
    // friend внутри шаблона: своя для каждого T, находится только через ADL
    friend bool operator==(const T& a, const T& b) { return a.equals(b); }
    friend bool operator!=(const T& a, const T& b) { return !(a == b); }
};

struct Vec2 : Comparable<Vec2> {
    float x, y;
    bool equals(const Vec2& o) const { return x == o.x && y == o.y; }
};
// operator== для Vec2 сгенерирован автоматически, в глобальном namespace его «нет»
```

Barton-Nackman / неправильно

```
// 1) свободный шаблонный оператор в общем заголовке утилит
template <class T>
bool operator==(const T& a, const T& b) { return a.equals(b); }
// теперь он КАНДИДАТ при сравнении любых двух объектов одного типа
// во всём проекте. Ошибки инстанцирования прилетают из мест,
// не имеющих к нему никакого отношения

// 2) дописать в std, чтобы «точно нашлось»
namespace std {
    bool operator==(const Vec2& a, const Vec2& b);   // UB
}
// специализировать шаблон std для своего типа можно
// добавлять в std новые перегрузки и функции нельзя!!!
```

```
class NAU_KERNEL_EXPORT IGenLoad
{
  virtual bool ceaseReading();
  // ....
};

class NAU_KERNEL_EXPORT LzmaLoadCB : public IGenLoad
{
public:
  void close();

  ~LzmaLoadCB()
  {
    close();
  }
  // ....
};

void LzmaLoadCB::close()
{
  if (isStarted && !isFinished && !inBufLeft && rdBufPos >= rdBufAvail)
    ceaseReading();
  // ....
```

V1053 Calling the 'ceaseReading' virtual function indirectly in the destructor may lead to unexpected result at runtime.

Мы уже нашли 4 похожие ошибки

Виртуальные вызовы в ctor/dtor

… / откуда взялась

Коварная ловушка C++

Не столько идиома, сколько набор обходных приёмов вокруг одной из самых коварных ловушек C++ — вызова виртуальной функции из конструктора или деструктора.

Поведение в конструкторе

Во время конструирования базового класса объект ещё «является» **только базой, производная часть не сконструирована.**

Виртуальный вызов из конструктора базы уйдёт в реализацию базы, даже если вы создаёте объект производного типа.

Поведение в деструкторе

В деструкторе зеркально: производная часть уже разрушена, и вызов уходит в реализацию базового класса.

Железная логика языка

Стандарт намеренно «опускает» динамический тип объекта до текущего конструируемого/разрушаемого класса. Вызвать переопределение в производном классе, чьи поля ещё не инициализированы или уже уничтожены, значило бы работать с мусором.

Совет Скотта Майерса

Проблему и её обходы детально разобрал **Скотт Майерс в Effective C++, выделив отдельный пункт: «никогда не вызывайте виртуальные функции при конструировании или разрушении».**

Calling Virtuals / в играх

Проблема в игровых иерархиях

UI-виджеты, компоненты, где возникает сильное желание настроить объект полиморфно прямо в конструкторе.

Индустриальный обходной путь

Игровая индустрия разработала встроенные архитектурные решения. Появился явный жизненный цикл объекта, как `BeginPlay` в Unreal или `OnEnable` / `Start` в Unity.

Двухфазная инициализация

Явная двухфазная инициализацию, отделяющую момент создания объекта от момента его полной готовности. Это позволяет безопасно выполнять виртуальные вызовы уже над полностью сконструированным объектом правильного типа.

Практика разработки игр

Тяжелая полиморфная инициализация почти никогда не выполняется в конструкторах игровых объектов. Вместо этого полагаются на коллбэки жизненного цикла, которые движок вызывает позднее.

Calling Virtuals / правильно

```
struct Widget {
    Widget() { /* НЕ звать здесь virtual draw() — уйдёт в Widget::draw */ }
    virtual void draw() { /* база */ }
    void init() { draw(); }   // безопасно: объект уже полностью сконструирован
};
struct Button : Widget { void draw() override { /* кнопка */ } };


// Двухфазная инициализация, чтобы получить полиморфизм при "создании":
auto b = std::make_unique<Button>();
b->init();   // вот теперь draw() уйдёт в Button::draw
```

Calling Virtuals / неправильно

```
struct Widget {
    Widget() {
        layout();               // уйдёт в Widget::layout, ВСЕГДА
    }
    virtual ~Widget() {
        detach();               // уйдёт в Widget::detach, ВСЕГДА
    }
    virtual void layout() { /* база */ }
    virtual void detach() { /* база */ }
};

struct ScrollView : Widget {
    void layout() override { /* никогда не вызовется из конструктора */ }
};
```

Единственное место в C++, где **virtual не виртуально.**

```
class CKeyMap {
public:
  CKeyMap();
  ~CKeyMap();
  ....
  virtual void swap(CKeyMap&);
  virtual void addHalfDuplexModifier(KeyID key);
  virtual void finish();
  ....
};

class CMockKeyMap : public CKeyMap { .... };

CKeyMap* m_keyMapPtr;

CKeyState::~CKeyState()
{
  if (m_keyMapPtr)
    delete m_keyMapPtr;
}
```

V599 The destructor was not declared as a virtual one, although the 'CKeyMap' class contains virtual functions.

Мы уже нашли 46 похожих ошибок

Virtual ctor

Virtual Constructor / откуда взялась

Фольклор плюсов, и строго говоря «деревянный как стекло»:

> конструктор в C++ **виртуальным быть не может, но**

`clone`

Задача «скопировать объект, зная только указатель на базовый класс».

Нельзя написать `new Derived(*basePtr)`, не зная, что это `Derived`, но можно объявить `clone()`, который каждый наследник переопределяет, возвращая копию себя своего точного типа.

`create`

Виртуальное создание нового объекта того же типа без копирования.

История

Подход популяризировал Клайн. Концептуально это знание восходит к ранним обсуждениям обхода невиртуальности конструкторов в C++.

Virtual Constructor / в играх

Виртуальное клонирование самая обычная рабочая лошадка в разработке игр:

Спавнеры

Для создания врагов и предметов, используя эталонный экземпляр для копирования.

Сохранение/Загрузка

В системах сохранения/загрузки, где необходимо воссоздать состояние объектов.

Редакторы

В инструментах и редакторах, когда нужно дублировать объекты по указателю на базовый класс.

Virtual Constructor / правильно

```
struct Enemy {
    virtual ~Enemy() = default;
    virtual std::unique_ptr<Enemy> clone() const = 0;   // виртуальное копирование
    virtual std::unique_ptr<Enemy> create() const = 0;  // виртуальное создание
};

struct Orc : Enemy {
    int rage;
    std::unique_ptr<Enemy> clone()  const override {
      return std::make_unique<Orc>(*this);
    }
    std::unique_ptr<Enemy> create() const override {
      return std::make_unique<Orc>();
    }
};

// Знаем только Enemy*, но получаем правильную копию правильного типа:
std::unique_ptr<Enemy> spawn_copy(const Enemy& prototype) {
  return prototype.clone();
}
```

Virtual Constructor / неправильно

Непереопределённый `clone()`

```
// 1) новый наследник — clone не переопределили
struct Berserker : Orc {
    float frenzy_timer;
    // clone() унаследован от Orc и вернёт... Orc
};
// спавнер честно клонирует прототип
// и получает орков вместо берсерков
// компилятор молчит: override просто отсутствует
```

Возврат сырого указателя

```
// 2) clone возвращает сырой указатель
virtual Enemy* clone() const;
// кто удаляет? когда?
```

Потеря состояния при клонировании

```
// 3) clone создаёт базу вместо себя
std::unique_ptr<Enemy> clone() const override {
    return std::make_unique<Orc>();
    // забыли *this копия без состояния
}
```

Чрезмерное использование в горячем коде

```
// 4) clone в хотпасе
// виртуальный вызов + аллокация
for (auto& e : particles)
    pool.push_back(e->clone());
    // на каждый партикл каждый кадр
```

ORCA SLICER

V553 The length of 'init_fff_params' function's body is more than 2000 lines long. You should consider refactoring the code.

Pimpl

Pimpl / откуда взялась

Что это?

Класс прячет все свои приватные члены за указателем на скрытую структуру реализации. В заголовке остаётся публичный интерфейс и единственное поле указатель на неполный тип `Impl`.

Зачем: Время компиляции

Изменение внутренних деталей класса больше не требует перекомпиляции всех, кто включает его заголовок. Э

Зачем: Стабильность ABI

Обеспечивает бинарную инкапсуляцию и стабильность ABI: размер и внутренняя раскладка класса остаются неизменными для внешнего кода. Это позволяет обновлять библиотеки без нарушения обратной совместимости.

История и имена

Концепция Handle/Body Джеймса Коплина (1992).

Известные названия: «Cheshire Cat» (Джон Каролан) и «Compilation Firewall» или собственно «Pimpl», популяризированные Хербом Саттером.

Pimpl / в играх

Изоляция и время сборки

Ускорения времени компиляции и изоляции тяжёлых сторонних зависимостей.

Где избегают

Хотпас, рендер.

Типичные сценарии

Идеально подходит для обёрток над сложными API.

Изоляция зависимостей

Помогает избежать "протекания" больших заголовков (Bullet, FMOD, SDK) в код всего игрового движка..

Pimpl / правильно

```
// physics_world.h заголовок ничего не знает о внутренностях
class PhysicsWorld {
public:
    PhysicsWorld();
    ~PhysicsWorld();                 // объявлен здесь, определён в .cpp
    void step(float dt);
private:
    struct Impl;                     // неполный тип
    std::unique_ptr<Impl> impl_;
};

// physics_world.cpp
struct PhysicsWorld::Impl {
    btDiscreteDynamicsWorld bullet;  // тяжёлый заголовок не утёк наружу
    std::vector<RigidBody> bodies;
};

PhysicsWorld::PhysicsWorld() : impl_(std::make_unique<Impl>()) {}
PhysicsWorld::~PhysicsWorld() = default;   // тут Impl полный checked delete доволен
void PhysicsWorld::step(float dt) { impl_->bullet.stepSimulation(dt); }
```

Pimpl / как неправильно

```
// 1) Pimpl на мелком значении в хотпасе
class Vector3 {
    struct Impl;
    std::unique_ptr<Impl> impl_;     // аллокация на каждый вектор
public:                              // и кеш-промах на каждое обращение к .x
    float x() const;
};
std::vector<Vector3> positions(100000);   // сто тысяч аллокаций
```

У идиомы четыре названия: Handle/Body, Cheshire Cat, Compilation Firewall и Pimpl.

Три из них красивые, и одно неправильное и прижилось, разумеется, четвёртое.

Fast Pimpl

Fast Pimpl / откуда взялась

Что это?

Устраняет главный недостаток Pimpl **динамическую аллокацию.**

Вместо указателя на кучу, `Impl` размещается **напрямую внутри объекта через буфер памяти и placement new.**

Выгода

Сохраняется инкапсуляция Pimpl, но убирается аллокация и лишняя индирекция, избегая кеш-промахов.

Объект становится **плоским в памяти, сочетая инкапсуляцию и производительность.**

История

Идиома детально разобрана **Хербом Саттером в "Exceptional C++" под названием «The Fast Pimpl Idiom».**

Fast Pimpl / где живёт в геймдеве

Типичные кандидаты

Обёртки над платформенно-зависимыми хендлами, сокеты, файловые дескрипторы, таймеры высокого разрешения.

Почему встречается редко

ручная синхронизация размера `Impl`

специфический профиль использования (горячий путь **плюс желание скрыть внутренности)**

Fast Pimpl / правильно

```
// fast_pimpl.h
class SoundEmitter {
public:
    SoundEmitter();
    ~SoundEmitter();
    void play();
private:
    struct Impl;
    static constexpr std::size_t kSize  = 64;   // подобрано под sizeof(Impl)
    static constexpr std::size_t kAlign = 16;
    alignas(kAlign) std::byte storage_[kSize];
    Impl* impl() { return reinterpret_cast<Impl*>(storage_); }
};

// fast_pimpl.cpp
struct SoundEmitter::Impl { /* поля микшера, голоса, фейды */ };
static_assert(sizeof(SoundEmitter::Impl)  <= 64,  "увеличь kSize");
static_assert(alignof(SoundEmitter::Impl) <= 16, "увеличь kAlign");
SoundEmitter::SoundEmitter()  { new (storage_) Impl(); }
SoundEmitter::~SoundEmitter() { impl()->~Impl(); }
```

Fast Pimpl / неправильно

Отсутствие проверок размера

```
// 1) без static_assert
class SoundEmitter {
    struct Impl;
    alignas(16) std::byte storage_[64];
};
// кто-то добавил поле в Impl, sizeof стал 72
// placement new пишет за границу буфера в соседние поля объекта
```

Непереносимый размер буфера

```
// 2) размер подобран под одну платформу
static constexpr std::size_t kSize = 64;
// померили на x64 Windows
// на другой платформе sizeof(std::mutex) внутри Impl другой
// и падает только та конфигурация, которую собирают раз в неделю
```

Забытое выравнивание

```
// 3) забыли выравнивание
std::byte storage_[64];                    // без alignas
// Impl содержит SIMD-поле, требующее 16 байт выравнивания
// на некоторых архитектурах это не «медленно», а «падает»
```

```
class UnicodeString
{  UnicodeString(const wchar_t *lpwszData)
  {
    SetEUS();
    Copy(lpwszData);
  }
  const wchar_t *CPtr() const { return m_pData->GetData(); }
  operator const wchar_t *() const { return m_pData->GetData(); }
  // ....
}
typedef UnicodeString FARString;
struct TreeItem
{
  FARString strName;
  // ....
}
TreeItem **ListData;
void TreeList::SetTitle()
{
  // ....
  if (GetFocus())
  {
    FARString strTitleDir(L"{");
    const wchar_t *Ptr = ListData
                         ? ListData[CurFile]->strName   // <=
                         : L"";                         // <=
```

V623 Consider inspecting the '?:' operator. A temporary object of the 'UnicodeString' type is being created and subsequently destroyed.

Мы уже нашли 91 похожую ошибку

```
class UnicodeString
{  UnicodeString(const wchar_t *lpwszData)
  {
    SetEUS();
    Copy(lpwszData);
  }
  const wchar_t *CPtr() const { return m_pData->GetData(); }
  operator const wchar_t *() const { return m_pData->GetData(); }
  // ....
}
typedef UnicodeString FARString;
struct TreeItem
{
  FARString strName;
  // ....
}
TreeItem **ListData;
void TreeList::SetTitle()
{
  // ....
  if (GetFocus())
  {
    FARString strTitleDir(L"{");
    const wchar_t *Ptr = ListData
                         ? ListData[CurFile]->strName.CPtr()
                         : L"";
```

V623 Consider inspecting the '?:' operator. A temporary object of the 'UnicodeString' type is being created and subsequently destroyed.

Мы уже нашли 91 похожую ошибку

Most Vexing Parse

```
struct Hit {
      Hit()  { cout << "default"; }
      Hit(int dmg) { cout << "damage"; }
}



int main() {
       Hit(x);
}
```

30 дней бесплатно
по промокоду

WEBINAR_IDIOMA

PVS-Studio

Заметки о ремесле,
программировании и
внутренностях игр

dalerank