Современные технологии разработки ПО

Современный C++

Содержание

- Контроль вариативности состояния
- Тёмная сторона и способы спасения
- RAII
- Интеллектуальные указатели
- Полиморфизм и C++
- Отсечение — Воландеморт вернулся

Часть 1
Контроль вариативности состояния

Типы вариативности состояний класса *

- **Неизменяемый (immutable)** — объект никогда не изменяет своего состояния
  - для таких объектов инвариант достаточно проверить в конструкторе
- **Копируемый** — при копировании объект не нарушает своего состояния:
  - при создании (конструктор копирования),
  - при передаче в качестве параметра методу (конструктор копирования),
  - при присваивании (оператор присваивания).
- **Некопируемый** — объект может быть только переносим без нарушения своего состояния

При ca

```cpp
#include <iostream>
#include <string>

using namespace std;

class Person
{
    string   _first_name;
    string   _last_name;
    uint16_t _year_of_birth;

public:
    Person() = delete;

    Person(string first_name, string last_name, uint16_t year_of_birth)
        : _first_name(first_name), _last_name(last_name), _year_of_birth(year_of_birth)
    {}

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

    const string & getFirstName() const { return _first_name; }
    const string & getLastName() const { return _last_name; }
    uint16_t getYearOfBirth() const { return _year_of_birth; }
};

int main(int , char **)
{
    Person citizen("Иван", "Иванов", 1993);

    cout << "Гражданин " << citizen.getFirstName() << " " << citizen.getLastName()
         << ", год рождения " << citizen.getYearOfBirth()
         << endl;

    return 0;
}
```

Пример предотвращения создания копии объекта в проекте на C++

```
class SqlDataCollector : public SymbolTable      padding size of 'miniDB::sql::SqlDataCollector'
{
    std::string                    _database_path; ///< Путь к файлам таблиц

    std::vector<DatasetState>      _from;          ///< Массив состояний наборов данных
    std::map<std::string,Dataset> _ds_set;         ///< Ассоциативный массив наборов данных
    DataKeeperPolicy               _policy;        ///< Политика работы с форматом хранения таблиц

public:
    SqlDataCollector() = delete;
    SqlDataCollector(const SqlDataCollector & ) = delete;
    SqlDataCollector operator=(const SqlDataCollector & ) = delete;

    /*!
     * \brief Конструктор класса управления наборами данных
     *
     * \param database_path       Путь к файлам таблиц
     * \param parent_symbol_table Указатель на вышестоящую таблицу имён
     * \param policy              Политика работы с форматом хранения таблиц
     */
    SqlDataCollector(const std::string & database_path, const SymbolTable * parent_symbol_table,
        : SymbolTable(parent_symbol_table)
        , _database_path(database_path)
          policy(policy)
```

Адаптивный код по состоянию объекта *

- Чем больше состояний у объекта (сложнее определение инварианта класса), тем меньше адаптивность кода
- Нужно минимизировать изменчивость состояния класса, в идеале до immutable-объектов (особенно важно для многопоточных и распределённых систем)
- Нужно проверять инвариант класса (см. **`std::assert`**) **после** манипуляций с состоянием класса (часто делается в начале методов — см. «контрактное программирование»)
- Нужно ограничивать возможность копирования, если это может нарушить инвариант класса
- Если вы затрудняетесь определить инвариант класса, значит есть повод пересмотреть структуру классов

Проверка инварианта класса в начале метода

```
bool Parser::parse(IStream & stream, ISyntaxTreeBuilder &builder)
{
    assert(!_g.parse_table.empty());
    assert(_g.first_compound_index < _g.columns.size());

    LexicalParameters param = makeLexicalParameters();

    Tokenizer tzer(convertToU16(_file_name), stream, param, 4);

    Token    t            = checkToken(tzer.getToken());
    size_t   column_no    = _g.getTerminalColumnIndex(t);
    bool     success      = true;
    bool     end          = false;
    uint16_t error_count = 0;
```

Часть 2
Тёмная сторона C++

Тёмная сторона C++, основные элементы *

Тёмная сторона	Светлая сторона
Указатели	Интеллектуальные указатели STL C++11: **`std::unique_ptr`**, **`std::shared_ptr`**, **`std::weak_ptr`**; передача по значению, ссылки
`new` / **`delete`**	Интеллектуальные указатели STL C++11, методы **`std::make_unique`** и **`std::make_shared`**
Ссылки	Интеллектуальные указатели STL C++11 (частично), передача по значению, семантика перемещения, ссылки
Массивы	Контейнерные классы STL C++11, ссылки
Оператор **`goto`**	Не используем.
Глобальные переменные	Объекты, передаваемые в параметрах, паттерн Одиночка
Макросы	Используем спецификаторы **`const`** и **`constexpr`** при объявлении переменной
Макрос **`NULL`**	Не используем. Ключевое слово **`nullptr`** в C++11

Тёмная сторона указателей и ссылок C++ *

- Может быть NULL (**`nullptr`**)
  - **крах программы**
- Можно забыть инициировать
  - **неопределённое поведение, крах программы**
- Может указывать на уже несуществующий объект
  - **неопределённое поведение, крах программы**
- Можно забыть выделить память
  - **неопределённое поведение, крах программы**
- Можно забыть освободить память
  - **утечка памяти**

Проблема висячих ссылок

- Ссылки лучше указателей
- Ссылки опасны, как и указатели
- Ссылки создают иллюзию безопасности, т. к. кажется, что мы работаем с объектом
  - (самого объекта может уже не быть)
- Лучше с ними не работать без крайней необходимости
- При работе со ссылками нужно применять «правила безопасности при работе со ссылками»

Правила безопасности при работе со ссылками (и указателями)

- Область видимости объекта должна быть больше, области видимости ссылки на него
- Объект не должен быть пересоздан за время работы ссылки:
  - (см. **`std::vector`**)
  - (особенно актуально при разработке многопотоковых приложений)
- Если объект «небольшой» и копируемый можно работать по значению, а не по ссылке
- Если объект временный, можно работать по значению (см. оптимизация и семантика перемещения)

Передача объектов по значению

- Объекты в C++ можно передавать по значению без специальных определений в его классе:
  - конструктора по-умолчанию,
  - конструктора копирования и
  - оператора копирования —
  - все они генерируются компилятором автоматически
- Исключение — ручное переопределение хотя бы одного такого элемента в классе (компилятор «пугается» и не генерирует остальные)
- Можно воспользоваться модификатором `default` для восстановления генерации этих элементов

Тёмная сторона копирования и передачи по значению

- Копирование неэффективно для крупных объектов
  - особенно, если они созданы только для передачи в качестве параметров в метод
- Но, есть есключение...
  - компилятор оптимизирует копирование, заменяя его перемещением для временных объектов (см. семантика перемещения)

Семантика перемещения в C++ 11

- Было до C++11:
  - lvalue (T) — сам объект;
  - rvalue (T&) — ссылка на объект;
  - конструктор копирования;
  - оператор присваивания.
- Добавлено в C++11:
  - rvalue-reference (T&&) — перемещающая ссылка;
  - конструктор перемещения;
  - оператор присваивания с перемещением;
  - встроенная функция std::move.
- Компилятор сам определит можно ли использовать перемещение, лучше не помогайте без крайней необходимости

Пример класса с перемещающей ссылкой

```
template<class T> class vector
{
    // ...
    // конструктор копирования
    vector(const vector&);
    // конструктор перемещения
    vector(vector&&);

    // обычное присваивание
    vector& operator=(const vector&);
    // оператор перемещения
    vector& operator=(vector&&);
};
// обратите внимание: конструктор и оператор перемещения
// принимают неконстантные &&
// они могут (и обычно делают) изменяют свои аргументы
```

Пример использования перемещающей ссылки

```
X a;
X f();

X& r1 = a;          // связывает r1 с a (lvalue)
X& r2 = f();        // ОШИБКА: rvalue

X&& rr1 = f();      // ok: связывает rr1 с временным объектом
X&& rr2 = a;        // ошибка: a – это lvalue
X&& rr3 = move(a);  // ok: rr3 заменяет a во владении
                    // move говорит компилятору, чтобы можно рассматривать переменную a
                    // в качестве rvalue
                    // move по сути конвертирует lvalue в rvalue
X b = move(a);      // ok: переменная a заменяется переменной b
                    // переменная a может стать недействительной
```

Часть 3
RAII

Зона видимости переменных в C++ *

- Удобно размещать переменные на стеке:

```
{
    // ...
    Person p;              // Начало зоны видимости p (конструктор p)
    bool ok = find(p);     // Использование p
    // ...
}                          // Конец зоны видимости p (деструктор)
```

- (C++ всегда освобождает переменные в обратном порядке от их объявления)

Распределение памяти в C++ *

- Если класс большой, содержит много полей, то на стеке лучше не размещать его:

```
{
    // ...
    std::unique_ptr<Person> p(new Person("Иван", "Иванов", 1993));
                                // Начало зоны видимости p (конструктор p)
    bool ok = find(*p); // Использование p
    // ...
    // delete p - не нужен!
}                               // Конец зоны видимости p (деструктор)
```

- Это и есть RAII

Идиома RAII *

- RAII — Resource Acquisition Is Initialization (бук. «получение ресурса есть инициализация»)
- RAII — Использование механизма контроля зоны видимости объекта для управления ресурсами, которые он содержит

Светлая сторона RAII

- RAII — мощнейший инструмент управления ресурсами для самых разных случаев:
  - системы реального времени (СРВ),
  - управление исключениями,
  - безопасное управление ресурсами,
  - просто удобно.
- RAII и сборка мусора

Тёмная сторона RAII

- Генерация исключений в деструкторе — плохо, т. к. деструктор может вызываться внутри исключения (в соответствии с идиомой RAII).
- Не надо увлекаться исключениями, т. к. они неуместны во многих окружениях:
  - встроенных системах,
  - драйверах и службах,
  - многопоточных приложениях,
  - в СРВ,
  - в высокопроизводительных программах, в т.ч. играх.

Часть 4
Интеллектуальные указатели

Интеллектуальные указатели *

- std::**unique_ptr** и std::**make_unique**
- std::**shared_ptr** и std::**make_shared**
- std::**weak_ptr**

std::unique_ptr *

- **std::unique_ptr** — умный указатель, который:
  - получает единоличное владение объектом через его указатель, и
  - разрушает объект через его указатель, когда unique_ptr выходит из области видимости.

Особенность unique_ptr *

- **unique_ptr** не может быть скопирован или задан через операцию присвоения
- Неконстантный **unique_ptr** может передать владение управляемым объектом другому указателю **unique_ptr**
- **const std::unique_ptr** не может быть передан, ограничивая время жизни управляемого объекта областью, в которой указатель был создан

Функция std::make_unique *

- Создаёт объект (с переменным количеством параметров в конструкторе) и оборачивает его в std::unique_ptr
- Обеспечивает безопасное выделение памяти под объект

```cpp
#include <iostream>
#include <memory>

struct Vec3
{
    int x, y, z;
    Vec3() : x(0), y(0), z(0) { }
    Vec3(int x, int y, int z) :x(x), y(y), z(z) { }
    friend std::ostream& operator<<(std::ostream& os, Vec3& v) {
        return os << '{' << "x:" << v.x << " y:" << v.y << " z:" << v.z  << '}';
    }
};

int main()
{
    // Use the default constructor.
    std::unique_ptr<Vec3> v1 = std::make_unique<Vec3>();
    // Use the constructor that matches these arguments
    std::unique_ptr<Vec3> v2 = std::make_unique<Vec3>(0, 1, 2);
    // Create a unique_ptr to an array of 5 elements
    std::unique_ptr<Vec3[]> v3 = std::make_unique<Vec3[]>(5);

    std::cout << "make_unique<Vec3>():      " << *v1 << '\n'
              << "make_unique<Vec3>(0,1,2): " << *v2 << '\n'
              << "make_unique<Vec3[]>(5):   " << '\n';
    for (int i = 0; i < 5; i++) {
        std::cout << "     " << v3[i] << '\n';
    }
}
```

std::shared_ptr *

- **std::shared_ptr** – умный указатель, с разделяемым владением объектом через его указатель:
  - несколько указателей **shared_ptr** могут владеть одним и тем же объектом;
  - объект будет уничтожен, когда последний **shared_ptr**, указывающий на него, будет уничтожен или покинет область видимости

Особенность shared_ptr *

- **shared_ptr** использует подсчёт ссылок для управления разделяемым ресурсом
- **shared_ptr** может не владеть ни одним объектом, в этом случае он называется пустым
- **shared_ptr** может хранить указатель на функцию
- **shared_ptr** является потокобезопасностым — несколько потоков могут одновременно считывать и записывать разные объекты **shared_ptr**, даже если они являются копиями с общим владельцем

shared_ptr

```
#include <memory>
using namespace std;

class F {};
class G : public F {};

int main()
{
    shared_ptr<G> sp0(new G);   // okay, template parameter G and argument G*
    shared_ptr<G> sp1(sp0);     // okay, template parameter G and argument shared_ptr<G>
    shared_ptr<F> sp2(new G);   // okay, G* convertible to F*
    shared_ptr<F> sp3(sp0);     // okay, template parameter F and argument shared_ptr<G>
    shared_ptr<F> sp4(sp2);     // okay, template parameter F and argument shared_ptr<F>
    shared_ptr<int> sp5(new G); // error, G* not convertible to int*     ◦ no matching con…
    shared_ptr<int> sp6(sp2);   // error, template parameter int and argument shared_ptr<F>
}
```

```cpp
#include <memory>
#include <string>

struct Song
{
    std::string artist;
    std::string title;

    Song(const std::string& artist_, const std::string& title_)
        : artist{ artist_ }, title{ title_ }
    {}
};
```

```cpp
using namespace std;

int main()
{
    // Используем функцию make_shared где только можем
    auto sp1 = make_shared<Song>("The Beatles", "Im Happy Just to Dance With You");

    // Теперь тоже самое, но с меньшей эффективностью - применение new для
    // конструирования объекта создаёт лишнюю безымянную переменную и выполняет копирование
    shared_ptr<Song> sp2(new Song("Lady Gaga", "Just Dance"));

    // Отдельно объявление и инициализация
    shared_ptr<Song> sp5(nullptr); // то же, что shared_ptr<Song> sp5;
    //...
    sp5 = make_shared<Song>("Elton John", "I'm Still Standing");

    // Инициализация через конструктор копирования, счётчик прибавляется
    auto sp3(sp2);

    // Инициализацяи через присваивание, счётчик прибавляется
    auto sp4 = sp2;

    // Обмен объекта с другим shared_ptr, sp1 и sp2
    // Вместе с объектами выполняется обмен их счётчиками
    sp1.swap(sp2);
}
```

std::weak_ptr

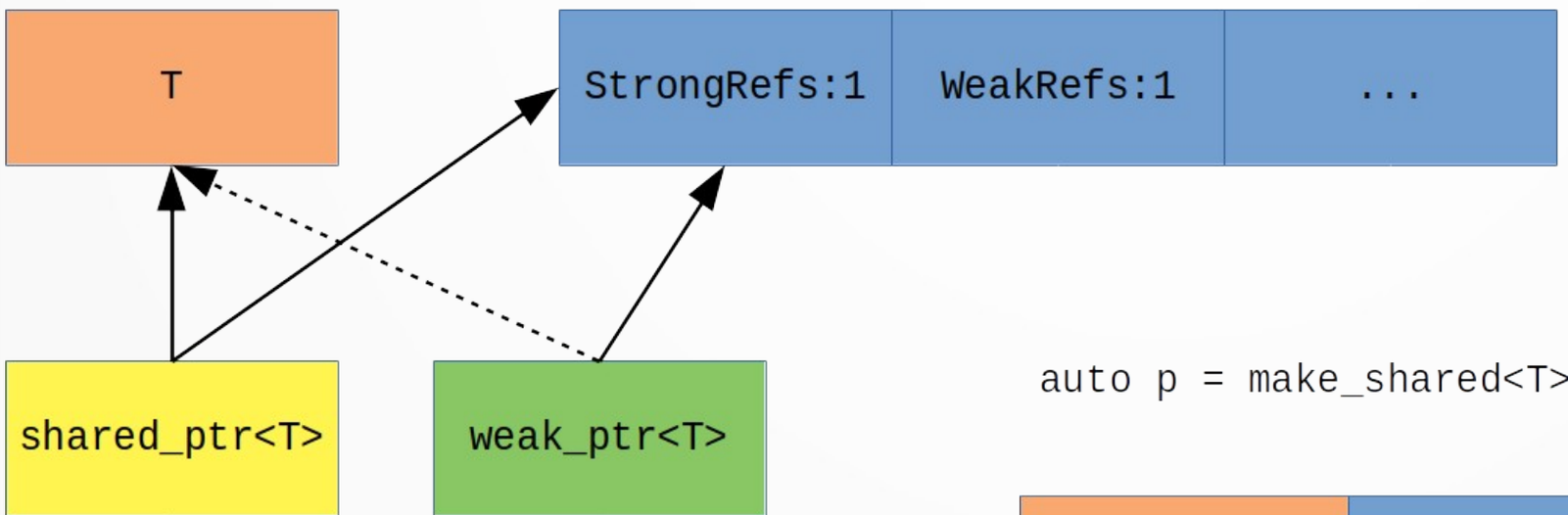
- **std::weak_ptr** — умный указатель, который содержит «слабую» ссылку на объект, управляемый указателем **std::shared_ptr**
- Чтобы получить доступ к управляемому объекту, указатель необходимо привести к типу **std::shared_ptr**
- **std::weak_ptr** моделирует временное владение: когда объект должен быть доступен только если он существует и может быть удален в любой момент

```cpp
#include <iostream>
#include <memory>

std::weak_ptr<int> gw;

void f()
{
    if (auto spt = gw.lock()) { // необходимо скопировать в shared_ptr перед использованием
        std::cout << *spt << "\n";
    } else {
        std::cout << "gw is expired\n";
    }
}

int main()
{
    {
        auto sp = std::make_shared<int>(42);
        gw = sp;

        f();
    }

    f();
}
```

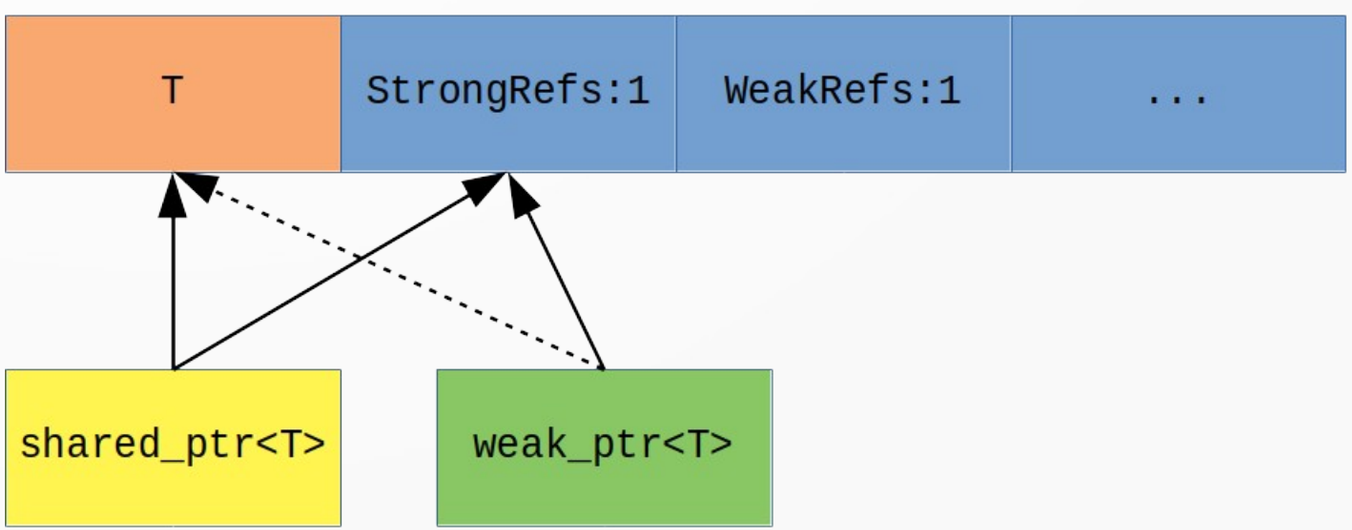
```
42
gw is expired
```

Схема работы с памятью shared_ptr, make_shared и weak_ptr

```
shared_ptr<T> p = new T(...);
```




```
auto p = make_shared<T>(...);
```



Часть 5
Полиморфизм и C++

Определения ООП *

- **Абстрактный класс** — класс, который имеет хотя бы один неопределенный («нулевой») виртуальный метод
- **Интерфейс** — абстрактный класс, у которого все его члены являются неопределенными («нулевыми») виртуальными методами
- **Полиморфные классы** — базовый класс с виртуальным методом (а также абстрактный базовый класс или интерфейс) и его производные классы, определяющие или переопределяющие виртуальные методы базового класса

Тёмная сторона виртуальных функций *

- Что было не так с виртуальными функциями?
  - можно ошибиться в названии
- Обязательное явное замещение виртуальных функций и финальность в C++11:
  - спецификатор описания функции **`override`**;
  - спецификатор описания функции **`final`**.

Пример применения спецификаторов override и final

```
struct B
{
    virtual void some_func();
    virtual void f(int);
    virtual void g() const;
};

struct D1 : public B
{
    void sone_func() override;           // ошибка: неверное имя функции
    void f(int) override;                // OK: замещает такую же функцию в базовом классе
    virtual void f(long) override;       // ошибка: несоответствие типа параметра
    virtual void f(int) const override; // ошибка: несоответствие cv-квалификации функции
    virtual int f(int) override;         // ошибка: несоответствие типа возврата
    virtual void g() const final;        // OK: замещает такую же функцию в базовом классе
    virtual void g(long);                // OK: новая виртуальная функция
};

struct D2 : D1
{
    virtual void g() const;              // ошибка: попытка замещения финальной функции
};
```

Часть 6
Отсечение — обратная сторона эффекти

Отсечение

- **Отсечение** — ситуация, когда при передаче по значению (копировании) объекта ошибочно используется базовый полиморфный класс, а инициализация данного объекта выполняется производным полиморфным классом. В этом случае, в место хранения не помещается дополнительные члены производного класса, в том числе таблица виртуальных методов (VMT).

Вопросы?