Объектно-ориентированное программирование

Object-oriented programming

IX. Контракты

Contracts

Выявление типа (type inference)

- Процесс **удовлетворения ограничений** (constraint satisfaction)
- Аппликация функции к аргументу генерирует ограничение, приравнивающее тип домена функции к типу аргумента
- Ограничения разрешаются методом исключения (unification)
  - Если решений нет – ошибка определения типа (overconstrained)
  - Если решений несколько, функция полиморфна, когда есть одно “лучшее” решение (underconstrained)
  - Если строго одно решение – проблема определения типа однозначна (uniquely determined)
- **Перегрузка операторов** создает дополнительные сложности в такой системе

http://www.cs.cmu.edu/~rwh/isml/book.pdf

Сравнение итераторов

```
const char * __begin1 = __range1;
const char * __end1 = __range1 + 10L;
for (; __begin1 != __end1; ++__begin1) {
    const char & c = *__begin1;
}
```

```
auto __begin1 = __range1.begin();
auto __end1 = __range1.end();
for (; ::operator!=(__begin1, __end1); __begin1.operator++())
{
    const char & c = __begin1.operator*();
}
```

Сравнение итераторов

```
template <typename _RandomAccessIterator>
void __reverse(_RandomAccessIterator __first,
               _RandomAccessIterator __last,
               std::random_access_iterator_tag) {
    if (__first == __last) return;
    --__last;
    while (__first < __last)
    {
        std::iter_swap(__first, __last);
        ++__first;
        --__last;
    }
}
```

Сравнение итераторов

```
class iterator {
    int *p{nullptr};
public:
    friend bool operator!=(iterator a, iterator b)
    { return a.p != b.p; }
    friend bool operator<(iterator a, iterator b)
    { return a.p < b.p; }
    ...
};
```

Сравнение итераторов

```
class iterator {
    int *p{nullptr};
public:
    friend bool operator!=(iterator a, iterator b)
    { return a.p != b.p; }
    friend bool operator<(iterator a, iterator b)
    { return *a.p < *b.p; }
    ...
};
```

Контракт – это …

Набор правил, устанавливаемых между вызывающей стороной и вызываемым поведением.

Например, для функции:

```
int f(double, double);
```

- явно не указана природа возвращаемого значения
- явно не указаны допустимые значения аргументов и отношения между ними

Контракты

- Пред-условие (ответственность “клиента”)

```
void set_by_ref(void *p[static 1], ...);
```

- Пост-условие (ответственность исполняющей стороны)

```
a_ = std::move(b_);
b_ = nullptr;
```

- Инварианта (условие, обязательное к соблюдению)

```
for (size_t i = 0; i < n; ++i) set_by_ref(a + i);
```

Инварианты класса

Определяют правильные состояния объекта конкретного класса, например:

- класс “вектор” предполагает, что **size**() **<= capacity**()

Инварианты класса допускается временно нарушать, пока выполняется метод класса, до тех пор, пока метод:

- не вернет результат
- не кинет исключение
- не вызовет **callback**-функцию
- не вызовет функцию, работа которой связана с состоянием объекта

Примеры

```
std::vector a {1, 2, 3, 4, 5};
...
std::vector b {std::move(a)};
a.push_back(); // ?
a.size();      // ?
a.clear();     // ?
```

Примеры

```
std::vector a {1, 2, 3, 4, 5};
...
std::vector b {std::move(a)};
a.push_back(); // нельзя вызывать на сброшенный объект
a.size();      // нельзя вызывать на сброшенный объект
a.clear();     // можно вызывать для приведения объекта в рабочее
               состояние
```

Как соблюдать контракты

- Использовать специальные библиотеки (**Boost.Contract** для **C++**)
- Использовать возможности языка (**assert**, **static_assert**, **concept** в **C++**)
- Подробно документировать код
- Написать собственный фреймворк для контрактов (например, через шаблоны или макро в **C++**)

https://www.boost.org/doc/libs/1_82_0/libs/contract/doc/html/index.html

X. Обработка ошибок

Error handling

Виды ошибок

- **Логические** ошибки (**bugs**) – результат неправильно написанного кода с точки зрения семантики (синтаксические ошибки не являются “багами”)
- Ошибки **исполнения** (**runtime**) – результат обработки неправильно сформированных данных
  - Восстанавливаемые ошибки
  - Невосстанавливаемые ошибки

Контракты необходимы для того, чтобы отделять **логические** ошибки от ошибок **исполнения**

Контракты показывают на потенциальные ошибки

Например, здесь явный баг:

```
std::vector v {1, 2, 3, 4, 5};
...
v.clear();
v.pop_back(); // не выполняется пред-условие
```

а здесь – ошибка исполнения:

```
auto load_file(size_t handle) {
    auto f = read(handle);
    return f; // не выполняется пост-условие
}
```

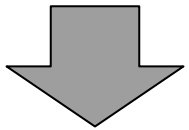
Ошибка – результат ошибочного ввода

- Системы подвержены сбоям
  - Влияние внешней среды
  - Преднамеренная атака на систему
  - “Железо” или “ПО” системы дефективно
  - Не предусмотренное дизайнером поведение пользователя
- Система должна продолжать работать в нормальном режиме

https://insights.sei.cmu.edu/documents/4054/2016_017_101_484207.pdf

Поведение системы в случае сбоя

- Fail-safe (переход в безопасный режим)
- Fail-soft (сохранение частичной функциональности)
- Fail-hard (приостановка работы)



- Обнаружение ошибки
- Восстановление до нормального состояния

Предотвращение возникновения ошибки

Например, деление на 0 или разыменование недействительного адреса:

```
int *p = calloc(10, sizeof(int));
for (size_t i = 0; i < 10; ++i)
    p[i] = x + i;
```

Из фонда лабораторных работ

```
int *p = calloc(10, sizeof(int));
// нет гарантии, что адрес выделен
if (NULL == p) {
/*1*/ printf("Error! Out of memory!");
/*2*/ return -1; // если внутри main
      // или
      fprintf(stderr, "Error! Out of memory!");
/*3*/ exit(-1);
}
for (size_t i = 0; i < 10; ++i)
   p[i] = x + i;
```

Приостановка работы программы

- Критически важный код может принять решение о приостановке
- Независимый код (application-independent) не может принимать решение о приостановке
  - **assert**(**int**)
  - **signal**(), **raise**()
  - **errno** (глобальный флаг, по умолчанию должен быть **0**)
  - локальные флаги (i.e. макро **EOF** или функция **feof**(**FILE***))
  - возвращаемые значения*

https://wiki.sei.cmu.edu/confluence/display/c/ERR02-C.+Avoid+in-band+error+indicators

Восстановление состояния

```
void str_build(std::string &s) {
    s += generate_prefix(); // может выйти заранее
    s += generate_suffix(); // может выйти заранее
}
```

Copy-and-swap

```
void str_build(std::string &s) {
    auto tmp = std::string{s};
    tmp += generate_prefix();
    tmp += generate_suffix();
    std::swap(tmp, s);
}
```

Такой подход гарантирует **атомарность** операции

assert и static_assert

```
size_t index(int a[n], int find) {
    for (size_t i = 0; i < n; i++) {
        if (find == a[i]) return i;
    }
    return ???;
}
```

assert и static_assert

```
size_t index(int a[n], int find) {
    size_t i = 0;
    for ( ; i < n; i++) {
        if (find == a[i]) return i;
    }
    assert (i != n);
}
```

assert и static_assert

```
size_t index(int a[n], int find) {
    size_t i = 0;
    for ( ; i < n; i++) {
        if (find == a[i]) return i;
    }
    assert (i != n);
}
```

assert используется не для того, чтобы выявлять “баги” в программе, а для того, чтобы указать на предположения, которые заложены в работу функции, вследствие чего они играют роль формализации условий контракта и в исходный код программы не включаются

assert и static_assert

```
int find(int a[n], int index) {
    if (index > 0 && index < n) return a[n];
    return ???;
}
```

Найдите ошибку в теле функции

```
FILE *f1, f2;
errno = NO_ERROR;
f1 = fopen(“1.txt”, “r”);
if(NULL == f1) return errno;
f2 = fopen(“2.txt”, “r”);
if(NULL == f2) return errno;
int *p = calloc(10, sizeof(int));
if(NULL == p) return errno;
fclose(f1);
fclose(f2);
free(p);
```

Найдите ошибку в теле функции

```
FILE *f1, f2;
errno = NO_ERROR;
f1 = fopen(“1.txt”, “r”);
if(NULL == f1) return errno;
f2 = fopen(“2.txt”, “r”);
if(NULL == f2) return errno;
int *p = calloc(10, sizeof(int));
if(NULL == p) return errno;
fclose(f1);
fclose(f2);
free(p);
```

Исключения (exceptions*)

```
size_t index(int a[n], int find) {
    for (size_t i = 0; i < n; i++) {
        if (find == a[i]) return i;
    }
    return n;
}
```

```
size_t index(int a[n], int find) {
    for (size_t i = 0; i < n; i++) {
        if (find == a[i]) return i;
    }
    throw std::out_of_range("Not found!");
}
```

Исключения

```
size_t m = index(a, n);
if (m == n) {
    perror("Not found!");
}
```

```
try {
    size_t m = index(a, n);
} catch(std::out_of_range &err) {
    std::cerr << err.what();
}
```

“We do not use **C++** exceptions”, **Google**

On their face, the benefits of using exceptions outweigh the costs, especially in new projects. However, for existing code, the **introduction of exceptions has implications on all dependent code**. If exceptions can be propagated beyond a new project, it also becomes **problematic to integrate the new project into existing exception-free code**. Because most existing **C++** code at **Google** is not prepared to deal with exceptions, it is comparatively difficult to adopt new code that generates exceptions.

https://google.github.io/styleguide/cppguide.html#Exceptions

Альтернативы исключениям - **std::optional**

```
std::optional<size_t> index(int a[n], int find) {
    for (size_t i = 0; i < n; i++) {
        if (find == a[i]) return i;
    }
    return {};
}
```

```
auto m = index(a, n);
if(!m.has_value())
    std::cerr << "Not found!";
else
    std::cout << *m;
```

Альтернативы исключениям - **expected**

```
std::pair<size_t, const char*> index(int a[n], int find) {
    for (size_t i = 0; i < n; i++) {
        if (find == a[i]) return {i, nullptr};
    }
    return {n, "Not found!"};
}
```

```
auto m = index(a, n);
if(m.second)
    std::cerr << m.second;
else
    std::cout << m.first;
```

Монада – это …

структура данных, которая включает в себя:

- единицу вычисления
- контейнер, изменяющий вычисляемое значение с помощью другого вычисления, который хранит полученный окончательный результат
- операцию приведения первоначального типа к типу контейнера
- операцию комбинирования функций, возвращающих значение этой структуры данных

Монада – это функтор

```
auto process(monad<std::string> in) {
    monad<std::string> out;
    int i = atoi(std::string(out));
    out = itoa(i * 2);
    return out;
}
```

```
auto process(monad<std::string> in) {
    return in
        .fmap(atoi)
        .fmap([](int i){return i * 2})
        .fmap(itoa);
}
```

Монада – это функтор

```
template <typename A>
struct monad {
    monad(const A&);

    template <typename B>
    auto fmap(function<B(A)> f) {
        return monad(f(*this));
    }
};
```

ranges уже это делает

```
std::vector<std::string> process(std::vector<std::string> in) {
    return in
        | ranges::view::transform(atoi)
        | ranges::view::transform([](int i){return i * 2})
        | ranges::view::transform(itoa)
        | ranges::to<std::vector>;
}
```

Монада – это моноид в категории эндофункторов

```
template <typename A>
struct monad {
    monad(const A&);

    template <typename B>
    auto fmap(function<B(A)> f) {
        return monad(f(*this));
    }

    template <typename B>
    auto bind(function<optional<B>(A)> f) {
        return join(fmap(f));
    }
};
```

Скалярное произведение векторов

```
(Insert +) ◦ (ApplyToAll *) ◦ Transpose
```

Например, для вектора {{1, 2, 3}, {6, 5, 4}}:

1. **Transpose** {{1, 2, 3}, {6, 5, 4}} : {{1, 6}, {2, 5}, {3, 4}}
2. (**ApplyToAll** *) {{1, 6}, {2, 5}, {3, 4}} : {6, 10, 12}
3. (**Insert** +) {6, 10, 12} : (**+** {6, (**+** {10, 12})})
4. **+** {6, 22} : 28

Альтернативы исключениям - **fmap** и **bind**

```
std::optional<size_t> m = index(a, n);
if(!m.has_value()) std::cerr << "Not found!";
else std::cout << *m;
```

```
auto m = index(a, n)
    .fmap( [](int i){ std::cout << i; } );
```

```
auto m = index(a, n)
    .bind( find ); // функция find может не вернуть
```

https://accu.org/journals/overload/26/143/brand_2462/

Монада – это …

структура данных, которая включает в себя:

- ~~единицу вычисления~~ функцию
- ~~контейнер, изменяющий вычисляемое значение с помощью другого вычисления, который хранит полученный окончательный результат~~ конструктор приведения типа
- ~~операцию приведения первоначального типа к типу контейнера~~ перегрузку **fmap** (transform в C++)
- ~~операцию комбинирования функций, возвращающих значение этой структуры данных~~ перегрузку **bind** (and_then в C++)

Более подробно о функторах и монадах в C++

https://www.youtube.com/watch?v=cE_YaFMhTK8