Объектно-ориентированное программирование

Object-oriented programming

IV. Обработка ошибок

Error handling

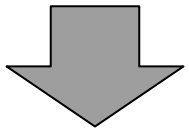
Ошибка - результат ошибочного ввода

- 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

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);
```

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; // нет гарантии, что адрес выделен
```

Из фонда лабораторных работ

```
int *p = calloc(10, sizeof(int));
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();
    s += generate_suffix();
}
```

Copy-and-swap

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

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

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;
```

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/

Более подробно о функторах и монадах в C++