Undefined behavior — конспект темы

Знакомство с неопределённым поведением

Суть **неопределённого поведения:** если программа нарушает правила языка, на её поведение не накладывается никаких ограничений.

Неопределённое поведение позволяет улучшить производительность программы:

- разрешает убирать из программы некоторые накладные расходы;
- разрешает выполнять оптимизации кода.

Эти функции принимают по ссылке `vector` и возвращают некоторый элемент массива. Одна использует оператор `[]`, а другая — метод `at`:

```
#include <vector>

int TestVectorIndexingOperator(std::vector<int>& v) {
    return v[3];
}

int TestVectorAt(std::vector<int>& v) {
    return v.at(3);
}
```

Первая функция быстрее. Когда индекс массива вышел за границы контейнера, случилось неопределённое поведение оператора `[]`. Это позволило избавиться от накладных расходов на проверку индекса.

Другие примеры:

- Повторный вызов оператора `delete` с одним и тем же значением указателя:

```
int main() {
    int *p = new int(42);
    delete p;
    delete p; // Неопределённое поведение: повторное удаление ранее удалённого объекта
}
```

- Вызов непарной версии оператора `delete`: удаление массива как одиночного объекта и наоборот:

```
int main() {
    int *p = new int(42);
    delete[] p; // Неопределённое поведение: одиночный объект удаляется как массив
}
```

- Одновременное чтение и запись одного и того же объекта в многопоточной программе без примитивов синхронизации:

```
#include <future>
#include <iostream>
#include <string_view>

using namespace std;

int main() {
    int i = 0;
    auto f = async([&i] {
        // Неопределённое поведение: чтение и запись переменной
        // производится в разных потоках без синхронизации
        ++i;
    });

    // Неопределённое поведение: чтение и запись переменной
    // производится в разных потоках без синхронизации
    ++i;
    f.get();
    cout << "i: "sv << i << endl;
}
```

Неожиданные последствия

Неопределённое поведение разработчику не друг. Из-за него программа может вести себя по-разному в разных версиях компилятора или с разными настройками компиляции.

Эта программа собирается и работает как надо при сборке компиляторами Visual Studio 2019 и Clang 10.0, но при сборке компилятором gcc 10.1 с ключом оптимизации `-O2` проявляется неопределённое поведение:

```
#include <iostream>
#include <string_view>

using namespace std;

const int NUM_PLANETS = 9;
const string_view PLANETS[] = {
    "Mercury"sv, "Venus"sv, "Earth"sv,
    "Mars"sv, "Jupiter"sv, "Saturn"sv,
    "Uranus"sv, "Neptune"sv, "Pluto"sv,
};

bool IsPlanet(string_view name) {
    for (int i = 0; i < NUM_PLANETS; ++i) {
        if (PLANETS[i] == name) {
            return true;
        }
    }
    return false;
}

void Test(string_view name) {
    cout << name << " is " << (IsPlanet(name) ? ""sv : "NOT "sv) << "a planet"sv << endl;
}

int main() {
    Test("Earth"sv);
    Test("Jupiter"sv);
    Test("Pluto"sv);
    Test("Moon"sv);
}
```

Так быть не должно. Проверяйте свой код, ищите и устраняйте причины неопределённого поведения.

Идиоматический C++ и неопределённое поведение

Идиоматический C++ помогает избегать проблем с неопределённым поведением.

В этом коде массив `PLANETS` и константа `NUM_PLANETS` — разные объекты:

```
bool IsPlanet(string_view name) {
    for (int i = 0; i < NUM_PLANETS; ++i) {
        if (PLANETS[i] == name) {
            return true;
        }
    }
```

```
    return false;
}
```

То же самое можно написать проще.

- Использовать функцию `std::size` для определения размера массива и отказаться от константы `NUM_PLANETS`:

```
bool IsPlanet(string_view name) {
    for (size_t i = 0; i < std::size(PLANETS); ++i) {
        if (PLANETS[i] == name) {
            return true;
        }
    }
    return false;
}
```

- Упростить цикл, используя range-based for:

```
bool IsPlanet(string_view name) {
    for (auto planet_name : PLANETS) {
        if (planet_name == name) {
            return true;
        }
    }
    return false;
}
```

- Заменить цикл на подходящий здесь алгоритм `std::find`:

```
#include <algorithm>
...
bool IsPlanet(string_view name) {
    return find(begin(PLANETS), end(PLANETS), name) != end(PLANETS);
}
```

- Применить алгоритм `std::ranges::find`, если ваш компилятор поддерживает C++20:

```
#include <algorithm>
...
```

```
using namespace std;
bool IsPlanet(string_view name) {
    return ranges::find(PLANETS, name) != end(PLANETS);
}
```

Идиоматический код лаконичнее, легче читается и меньше подвержен ошибкам, которые приводят к неопределённому поведению.

Инструменты для обнаружения неопределённого поведения

Статический анализатор Clang-tidy

Анализирует код программы и выдаёт предупреждения о реальных и потенциальных ошибках в коде, таких как:

- неопределённое поведение,
- ошибки copy-paste,
- некорректное использование библиотечных функций,
- использование устаревших языковых конструкций или библиотечных функций.

Clang-Tidy входит в состав LLVM вместе с компилятором Clang.

Отладочный режим стандартной библиотеки

Включается добавлением следующих опций командной строки:

- libstdc++: `-D_GLIBCXX_DEBUG` (обычно компилятор gcc),
- libc++: `-D_LIBCPP_DEBUG=1` (обычно компилятор clang),
- компилятор Visual C++: `-D_ITERATOR_DEBUG_LEVEL=2`.

Undefined behavior sanitizer

Специальный режим работы компилятора, который добавляет в скомпилированный код дополнительные проверки. Они ищут неопределённое поведение:

- использование нулевого указателя или указателя с неправильным выравниванием;
- переполнение в операциях над целыми числами со знаком;
- целочисленное деление на ноль.

Проверки охватывают код всей программы, а не только код стандартной библиотеки.