Безопасная работа с памятью в Rust

Конференция «Программирование и вычислительная математика», посвящённая 100-летию со дня рождения Николая Павловича Трифонова

Зиновкин А. И; Полякова И. Н (Кафедра Алгоритмических языков ВМК МГУ имени М.В Ломоносова)

Язык Rust

Разработка началась в 2006 году Грэйдоном Хором

- Первая стабильная версия вышла в 2015 году
- С 2009 года проект поддерживался Mozilla, сегодня развивается Rust Foundation

Ключевая идея

- • Безопасность памяти без сборщика мусора
- • Высокая производительность на уровне C и C++

Корректность сравнения Rust и C++

Сравнение Rust с языками, предоставляющими полный доступ к памяти, такими как C++, является **оправданным**.

Rust ограничивает прямое управление памятью в безопасной части, но через механизм *unsafe* позволяет выполнять низкоуровневые операции.

Rust и философия безопасной инкапсуляции

- Rust строит систему вокруг принципа: небезопасные операции должны быть скрыты внутри надёжных абстракций.

Формальное подтверждение

Исследование Nima Rahimi Foroushaani и Bart Jacobs.

Даже код с *unsafe* может быть доказан безопасным при модульной структуре и корректной инкапсуляции.

Сравнение Rust и C++ в аспекте безопасности

	C++	**Rust**
Гарантии управления памятью	Нет встроенных гарантий	Строгие гарантии на уровне компиляции
Механизм контроля	Ручное управление	Система владения и заимствования
Низкоуровневый доступ	Доступ напрямую, без ограничений	Через блоки unsafe строго ограниченные и контролируемые

Висячие указатели

Висячий указатель— это указатель, который ссылается на область памяти, которая была освобождена или больше не является действительной.

C++	Rust

int* createInt() {
int value = 42;
int* ptr = &value;
return ptr;
// Возврат указателя на локальную переменную — проблема!}
	fn create_int() -> &i32 {
	let x = 42;
	&x // Ошибка! 'x' выходит из области видимости
	}

Разыменование нулевых указателей

В C/C++ разыменование нулевых указателей — это известная проблема.

Попытка доступа к памяти через нулевой указатель приводит к аварийному завершению (`segmentation fault`) или непредсказуемым последствиям.

Так же данная уязвимость может быть использована для:

- Перезаписи указателей на функции;
- Перехода к определённым участкам памяти;
- Выполнения произвольного кода.

Тип Option в Rust

В Rust для явного представления отсутствующих значений используется тип `Option`.

Главные преимущества:

- Исключение разыменования нулевых указателей
- Принудительная обработка всех случаев
- Безопасность без потери производительности
- Option реализован как нуль-оптимизированное перечисление
- В рантайме занимает столько же памяти, сколько nullable-указатель в C++, но с гарантиями безопасности

Пример уязвимости с нулевым указателем

C++	Rust

int unsafe_length(const std::string* s) {
return s->length();
// UB если s == nullptr
}
	fn safe_length(s: Option<&String>) -> usize {
	s.map(|x| x.len()).unwrap_or(0)
	}

Сравнение Rust и C++: управление памятью

C++	Rust
Безопасность владения достигается через «умные» указатели	Модель владения встроена в язык
«Умные» указатели — библиотечные абстракции, не часть типовой системы	Применяется ко всем значениям по умолчанию
Их использование зависит от дисциплины программиста	Обеспечивает единообразие и предсказуемость поведения

Дополнительные инструменты безопасности

C++	Rust
Широкий набор внешних инструментов анализа и отладки (Valgrind, AddressSanitizer, статические анализаторы)	Контроль памяти встроен в типовую систему и компилятор(так же поддержка Valgrind,Miri)
Инструменты выявляют и смягчают последствия уже допущенных ошибок	Ошибки предотвращаются ещё на этапе компиляции
Их использоа Безопасность зависит от использования сторонних средств вание зависит от дисциплины программиста	Архитектурные гарантии встроены в язык

Заключение

Rust обеспечивает высокий уровень безопасности памяти благодаря встроенной системе владения и контроля времени жизни.

В отличие от C++, где многое зависит от дисциплины и внешних инструментов, Rust предотвращает ошибки уже на этапе компиляции.

Такой подход делает язык надёжным для критически важных приложений,

сохраняя привычную императивную парадигму.