C++ und Performance

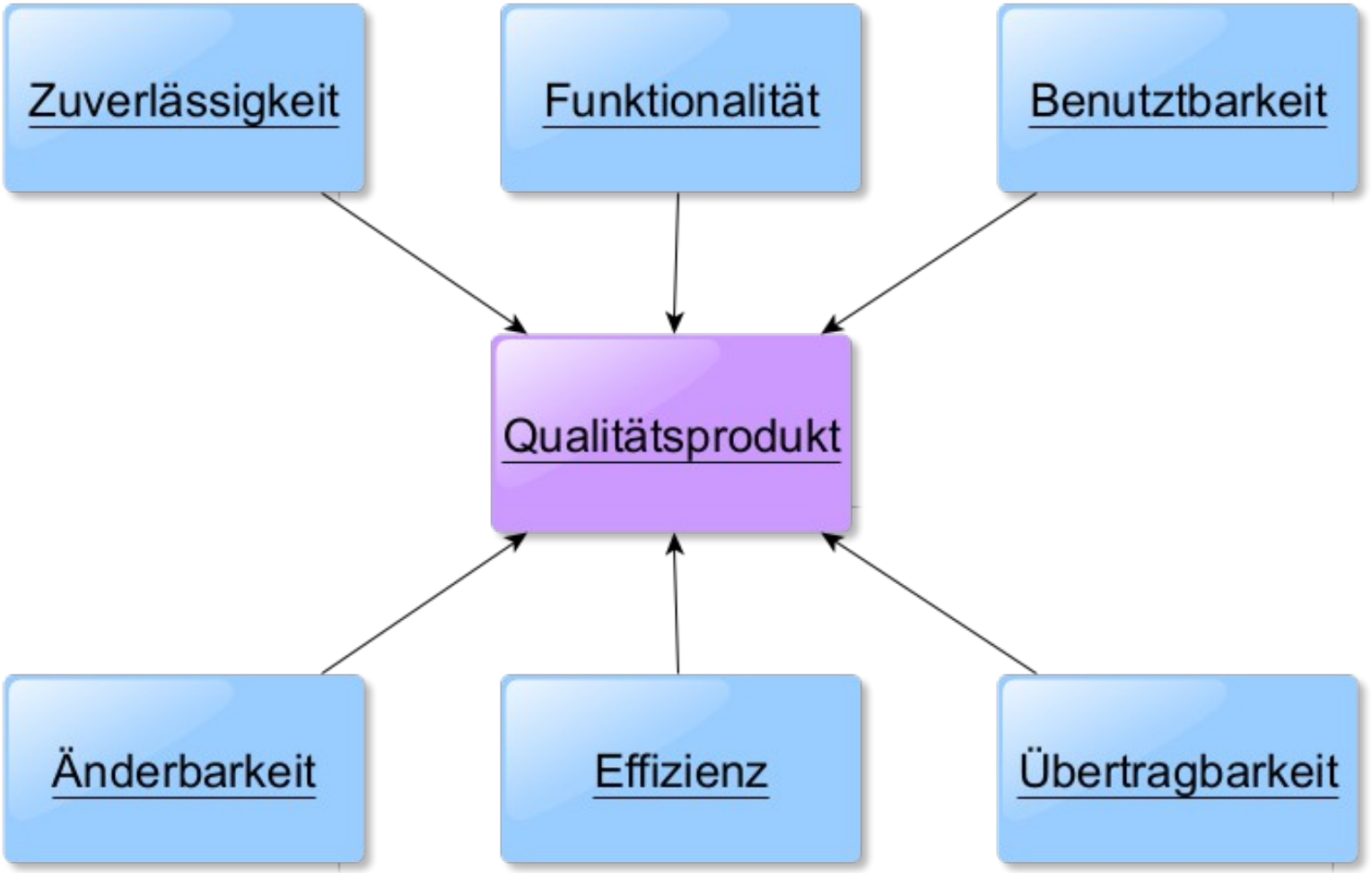
William Mendat
Angewandte Informatik

Inhaltsverzeichnis

Einleitung

- Das Upgrade zu C
- Fluch und Segen zugleich
- Viel mehr Kontrolle
- Zero-Overhead Prinzip
- Nahe Kopplung zur Hardware
- Vorteile einer hohen Programmiersprache

Take-Four-Problem



Templates - 1

- Sehr mächtig
- Dynamische Typen
- Instanziierung zur Kompilierzeit
- Kein Overhead bei nicht Verwendung
- Unleserlich und wirre Fehlermeldung.

Templates - 2

```
template <uint32_t Base>
struct Sum
{
	static const uint64_t result =
		Base + Sum<Base - 1>::result;
};

template<>
struct Sum<1>
{
	static const uint64_t result = 1;
};

int main(int argc, char** argv)
{
	std::cout << Sum<5>::result << std::endl;
	std::cin.get();
}
```

Constant Expressions

- Veröffentlicht mit C++ 11
- Keyword constexpr
- Leserlicher Code

```
constexpr uint64_t SumConstExpr(const uint32_t value) {
	return value <= 1 ? 1 : (value + SumConstExpr(value - 1));
}

int main(int argc, char** argv)
{
	std::cout << SumConstExpr(5) << std::endl;

	std::cin.get();
}
```

Speicher Management - 1

- Speicher – Wichtigste Komponente
- Stack vs Heap
- Programmierer entscheidet
- Anforderung von frischem Speicher
- Cache Misses

```
int main(int argc, char** argv)
{
	uint32_t** arr2d = new uint32_t * [5];
	for (uint16_t i{}; i < 5; ++i)
		arr2d[i] = new uint32_t[5];
}
```

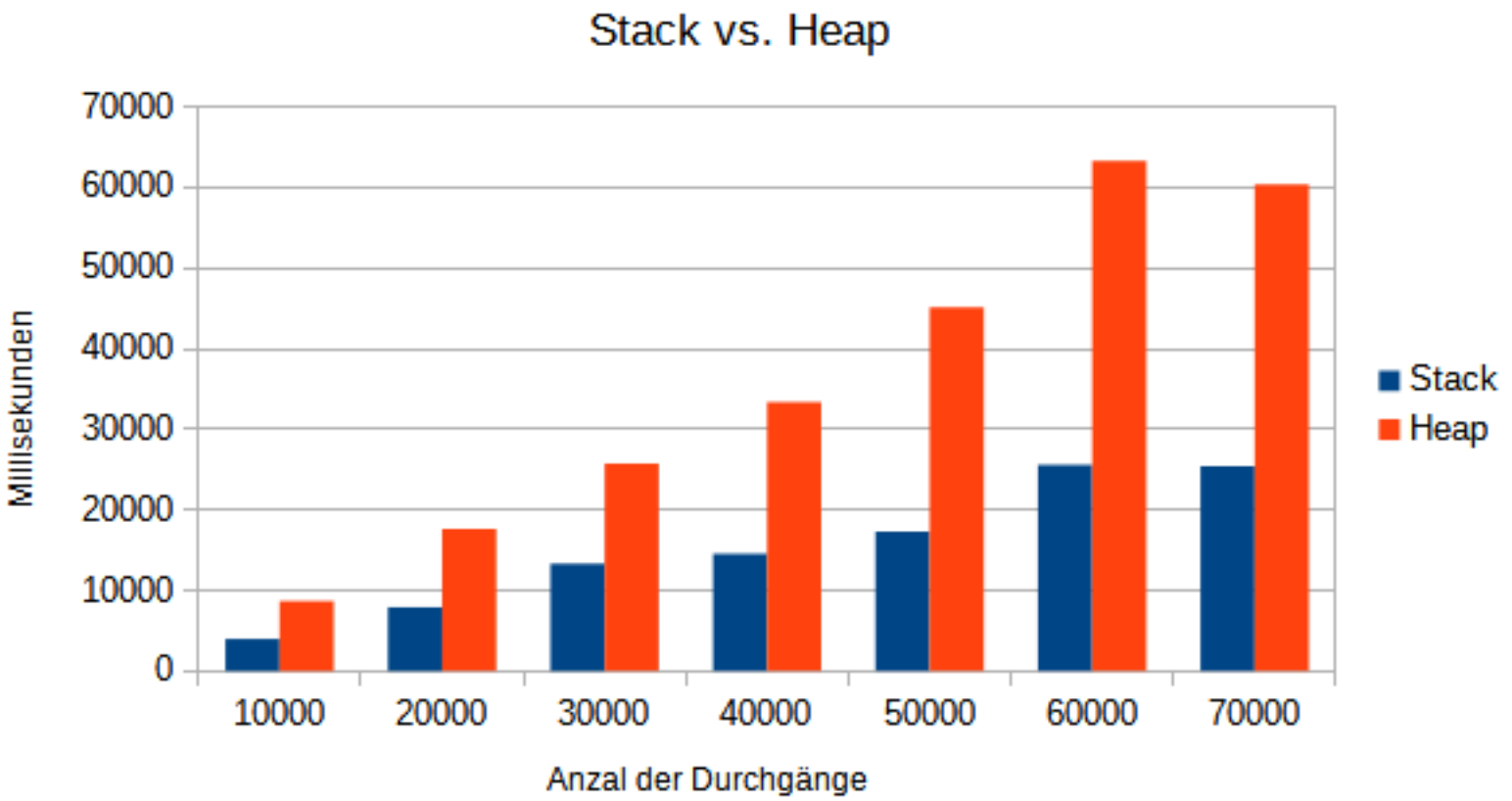
arr2d[0]: 010B5188
arr2d[1]: 010B51C8
arr2d[2]: 010B52C8
arr2d[3]: 010B5308
arr2d[4]: 010BEE78

Speicher Managment - 2

Speicher auf dem Heap zuweisen:

- Muss Multithreading Unterstützen
- Muss Synchronisiert werden
- Effizient verschiedene Größen verwalten
- Darf nicht über die Zeit verschlechtern
- Muss teils zum Kernel runter

Speicher Management - 3



Speicher Management - 4

- Heap Zuweisungen sind nicht Trivial
- Standard Implementation
- Eigen Implementation
- Externe Implementationen
- Tcmalloc, Mimalloc, Jmalloc, etc...
- Wann immer möglich Stack benutzen

Temporäre Objekte

- ***lvalues*** und ***rvalues***
- Werden vom Compiler erzeugt
- Erzeugt und Verworfen
- Dienen nur um Daten zu geben

```
String(const char* string) {
        m_Size = strlen(string);
        m_Data = new char[m_Size + 1];
        memcpy(m_Data, string, m_Size);
        m_Data[m_Size] = 0;
}

class Person{
public:
Person(const String& name)
        : m_Name(name) { }
}
```

Kopieren von Objekten

- Flache vs Tiefe Kopie
- Sehr teuer bei Komplexen Objekten
- Meinst nicht vonnöten
- Kopier Konstructor
- Heap Allokierung
- „Always pass by Const Reference“

```
String(const String& other) {
    m_Size = other.m_Size;
    m_Data = new char[m_Size + 1];
    memcpy(m_Data, other.m_Data, m_Size + 1);
}
```

Runtime Type Identification

- Sicheres *Casten*
- Metainformationen einer Klasse
- *dynamic_cast*
- Unnötiger Overhead
- Überprüfung zur Laufzeit
- Sofern möglich – Ausschalten
- Alternative: Virtuelle Funktionen

Fehlerbehandlungen - 1

- Stehen Performance im Weg
- Take-Four-Problem
- Laufzeit Überprüfungen
- Nicht Relevant für die Logik

Fehlerbehandlungen - 2

- Status Code vs. Exceptions
- Exceptions bieten einige Vorteile
- Sehr schlecht für Performance
- Verletzung des Zero-Overhead Prinzips
- Dynamische Erzeugung auf dem Heap
- Dynamische Ermittlung
- Keine Deterministische Zeitermittlung möglich

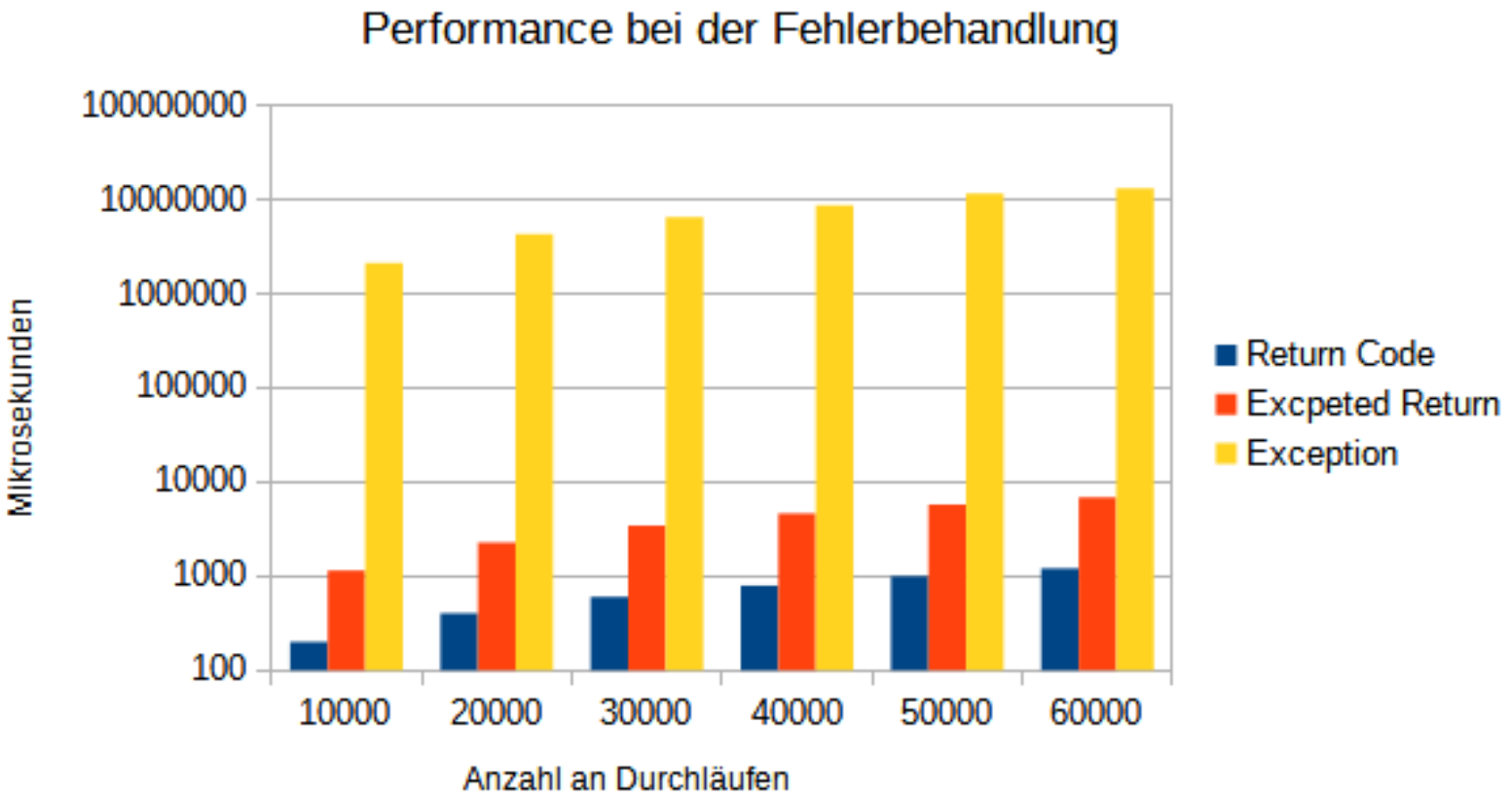
Fehlerbehandlungen - 3

- Kompromiss zwischen Exceptions und Status Codes
- Template Klasse „Expected“
- Keine Verletzung des Zero-Overhead Prinzips

```
template <class T>
class Expected {
private:
        union {
                T value;
                std::exception_ptr exception;
        };

        [...]
};
```

Fehlerbehandlungen - 4



Probleme mit Strings

- Behandlung wie Primitive Datentypen
- Heap Allokiert
- Operationen verursachen Heap allokierungen
- Verursachen viele Kopien

```
void* operator new(size_t size) {
        ++Allocations;
        return malloc(size);
}

int main(int argc, char** argv)
{
        String demo = String("Hello") + " World" + "!";
        std::cout << "New called " << Allocations <<
                "Times" << std::endl;
}
```

String Optimierungen - 1

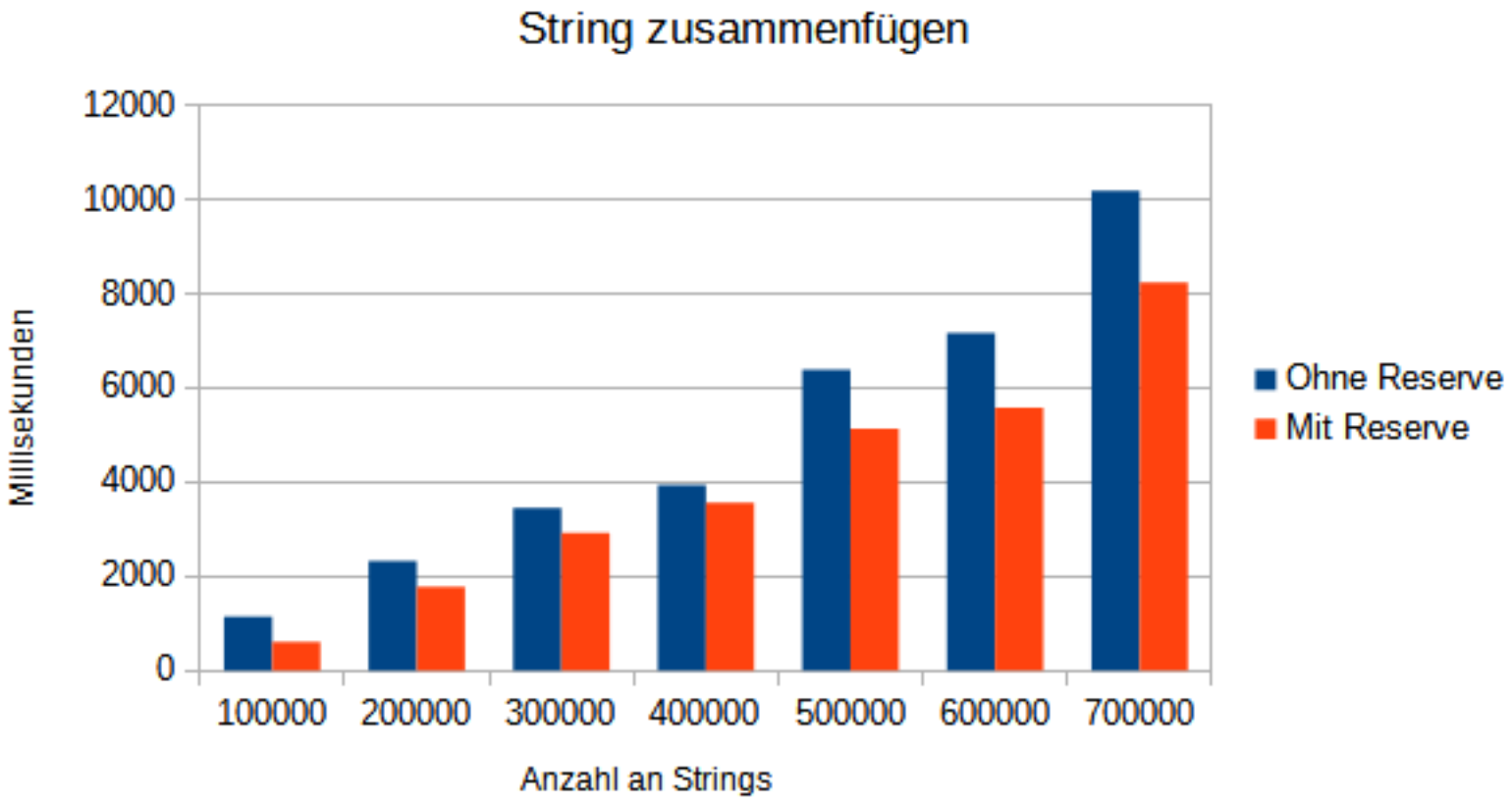
- Small String Optimization
- Veröffentlicht in C++ 11
- Keine Heap Allokierung
- Muss nichts extra gemacht werden

String Optimierungen - 2

- std::string_view
- Veröffentlicht mit C++ 17
- Ermöglicht „Einblick“ im String
- Keine zusätzliche Heap Allokierung

String Optimierungen - 3

Reserve:



Move-Semantiken

- Daten werden „Geklaut“
- Keine zusätzliche Heap Allokierung
- Viel mehr Kontrolle
- Move Konstruktor
- rvalue-refernce

```
class String {
    String(String&& other) {
        m_Size = other.m_Size;
        m_Data = other.m_Data;
        other.m_Data = nullptr;
        other.m_Size = 0;
    }
};
```

Demo

Fazit