(Advanced) C++ Design Patterns

C++ User Group Frankfurt

Kai Wolf, M.Sc.
mail@kai-wolf.me

Kai Wolf - SW Consulting

08.01.2021

Introduction

- Computer scientist (about 13 years work experience) working as an independent Software Consultant in last years
- Mainly in the following areas
  - Computer Vision / Image Processing
  - Machine Learning / Deep Learning
  - Mobile Computing / Embedded
  - Software Architecture / Build Systems / DevOps
  - Performance Optimizations
- Clients coming from a wide range of sectors such as
  - Automotive
  - Technology provider
  - Medicine
  - Startups
  - B2B, Software companies in general

Outline of this talk

- Assuming the audience is aware of the GoF Design Patterns

Outline of this talk

- Assuming the audience is aware of the GoF Design Patterns
  - Observer, Command, Visitor (Hey compiler hackers), ...

Outline of this talk

- Assuming the audience is aware of the GoF Design Patterns
  - Observer, Command, Visitor (Hey compiler hackers), ...
- No **in-depth** review of existing/well-established solutions

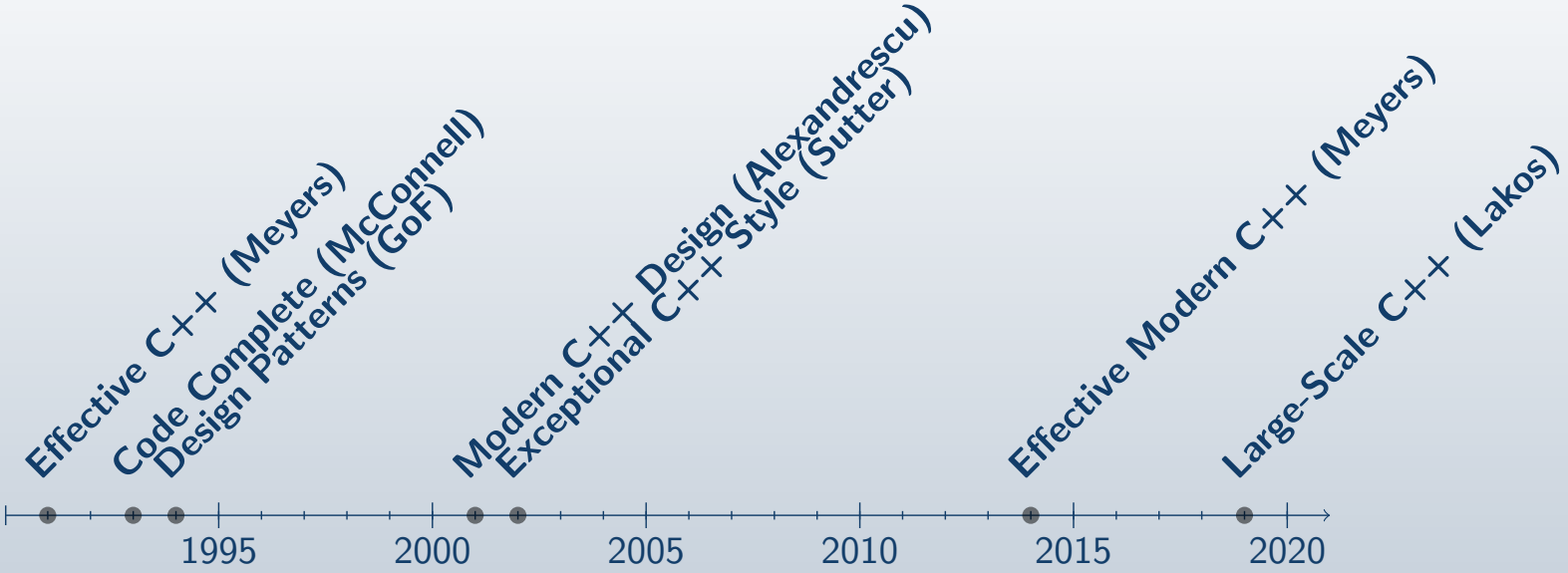
Outline of this talk

- Assuming the audience is aware of the GoF Design Patterns
  - Observer, Command, Visitor (Hey compiler hackers), ...
- No **in-depth** review of existing/well-established solutions
- Rather an overview with applicable examples

Outline of this talk

- Assuming the audience is aware of the GoF Design Patterns
  - Observer, Command, Visitor (Hey compiler hackers), ...
- No **in-depth** review of existing/well-established solutions
- Rather an overview with applicable examples
- Approaches that seem to work in production code (based on my project experience)

Once upon a time



Preamble

- Software design takes **data** and **computation** into account

Preamble

- Software design takes **data** and **computation** into account
- Specifically, we model our data into concrete types and apply operations on them

Preamble

- Software design takes **data** and **computation** into account
- Specifically, we model our data into concrete types and apply operations on them
- Design patterns describe working, reusable solutions to common software programming problems

Preamble

- Software design takes **data** and **computation** into account
- Specifically, we model our data into concrete types and apply operations on them
- Design patterns describe working, reusable solutions to common software programming problems
- There exist patterns for recurring problems on different parts of software development

Preamble

- Software design takes **data** and **computation** into account
- Specifically, we model our data into concrete types and apply operations on them
- Design patterns describe working, reusable solutions to common software programming problems
- There exist patterns for recurring problems on different parts of software development
  - resource management, container, concurrency, architecture, distribution, ...

Preamble

- Software design takes **data** and **computation** into account
- Specifically, we model our data into concrete types and apply operations on them
- Design patterns describe working, reusable solutions to common software programming problems
- There exist patterns for recurring problems on different parts of software development
  - resource management, container, concurrency, architecture, distribution, ...
- This talk will only cover some topics on interface design and modularization

Data Type

- Types are our friends in statically typed languages

Data Type

- Types are our friends in statically typed languages
- Static typing is a good thing (in C and C++)

Data Type

- Types are our friends in statically typed languages
- Static typing is a good thing (in C and C++)
- What exactly *is* a data type?

Data Type

- Types are our friends in statically typed languages
- Static typing is a good thing (in C and C++)
- What exactly *is* a data type?
  - a set of compile-time properties for an object:

Data Type

- Types are our friends in statically typed languages
- Static typing is a good thing (in C and C++)
- What exactly *is* a data type?
  - a set of compile-time properties for an object:
    - size and alignment

Data Type

- Types are our friends in statically typed languages
- Static typing is a good thing (in C and C++)
- What exactly *is* a data type?
  - a set of compile-time properties for an object:
    - size and alignment
    - set of valid values

Data Type

- Types are our friends in statically typed languages
- Static typing is a good thing (in C and C++)
- What exactly *is* a data type?
  - a set of compile-time properties for an object:
    - size and alignment
    - set of valid values
    - set of permitted operations

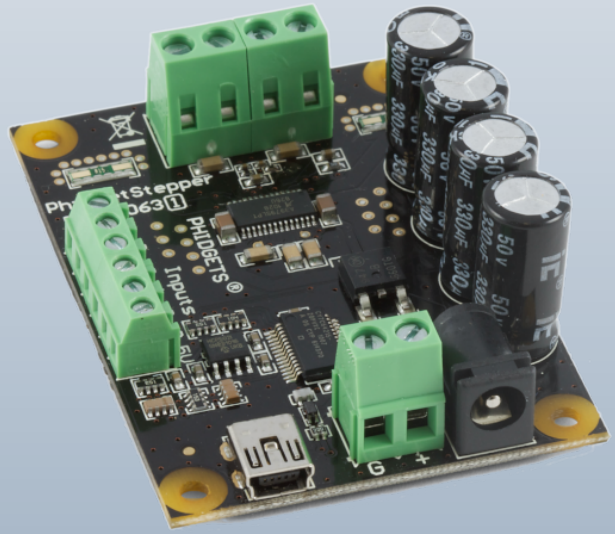
Data Type

- Types are our friends in statically typed languages
- Static typing is a good thing (in C and C++)
- What exactly *is* a data type?
  - a set of compile-time properties for an object:
    - size and alignment
    - set of valid values
    - set of permitted operations
- We can *exploit* this for constraining our software system under development

Motivating example

Context

In embedded development, hardware is typically controlled (nowadays) via so called memory-mapped devices. Hardware is controlled by flipping bits (register) on certain memory addresses.



Motivating example

- Those register addresses typically share a common basic type such as

```
using device_reg = unsigned volatile;
```

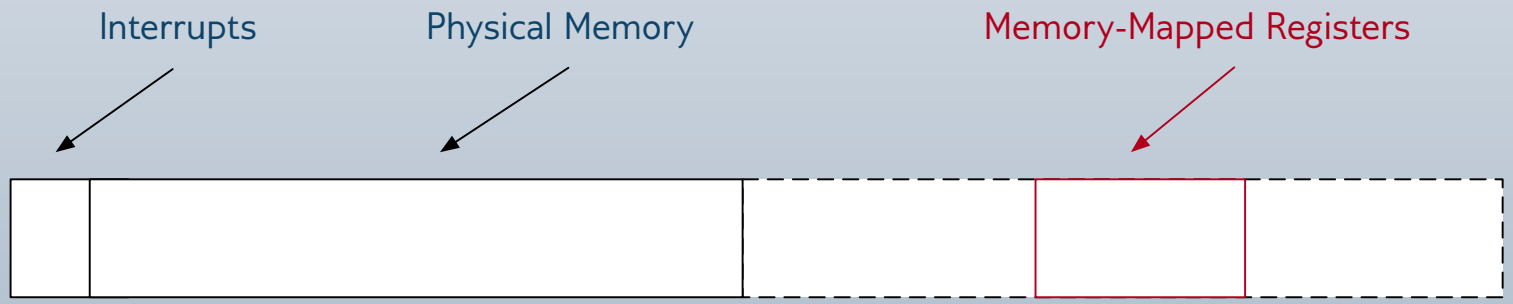
Motivating example

- Those register addresses typically share a common basic type such as

```
using device_reg = unsigned volatile;
```

- Addressing such register usually looks like this

```
#define STROBE1 ((device_reg *)0x1500)
#define STROBE2 ((device_reg *)0x1504)
```



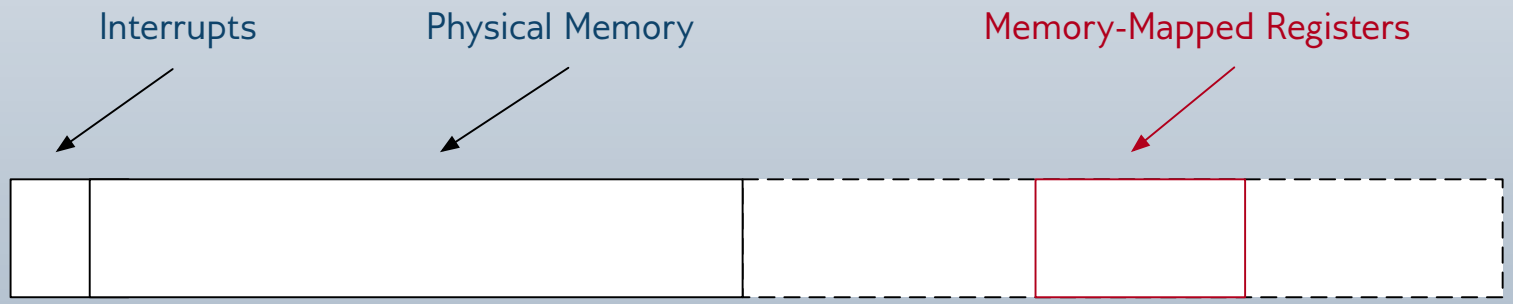
Motivating example

- Those register addresses typically share a common basic type such as

```
using device_reg = unsigned volatile;
```

- Addressing such register usually looks like this

```
#define STROBE1 ((device_reg *)0x1500)
#define STROBE2 ((device_reg *)0x1504)
```



- Modifying the hardware consists of bit shuffling on this particular address:

```
*STROBE1 |= 3;
```

Motivating example (cont.)

- This is prone to errors:

Motivating example (cont.)

- This is prone to errors:
  - Often there are registers that are **only** supposed to be read from

Motivating example (cont.)

- This is prone to errors:
  - Often there are registers that are **only** supposed to be read from
  - While other registers are supposed to be write-only

Motivating example (cont.)

- This is prone to errors:
  - Often there are registers that are **only** supposed to be read from
  - While other registers are supposed to be write-only
  - False read/writes can be a happy source of debugging

Motivating example (cont.)

- This is prone to errors:
  - Often there are registers that are **only** supposed to be read from
  - While other registers are supposed to be write-only
  - False read/writes can be a happy source of debugging
- We want to reduce the number of potential errors by making use of the type system to constrain access to those memory-mapped devices

Motivating example

```
template <typename t>
class WriteOnly {
  public:
    WriteOnly() = default;
    WriteOnly(const t &val) : val_(val) {}
    void operator=(const t &val) { val_ = val; }
    WriteOnly(const WriteOnly &) = delete;
    WriteOnly &operator=(const WriteOnly &) = delete;

  private:
    t val_;
};
```

[0] code::dive conference 2015 - Dan Saks - Sooner rather than later

Motivating example

```
template <typename t>
class WriteOnly {
  public:
    WriteOnly() = default;
    WriteOnly(const t &val) : val_(val) {}
    void operator=(const t &val) { val_ = val; }
    WriteOnly(const WriteOnly &) = delete;
    WriteOnly &operator=(const WriteOnly &) = delete;

  private:
    t val_;
};
```

Works as advertised

```
WriteOnly<int> strobe(17);
strobe = 23; // assignment works
int foo = strobe; // does not compile
```

[0] code::dive conference 2015 - Dan Saks - Sooner rather than later

Motivating example

```
template <typename t>
class ReadOnly {
  public:
    ReadOnly() = default;
    ReadOnly(const t &val) : val_(val) {}
    operator const t &() const { return val_; }
    const t *operator&() const { return &val_; }
    ReadOnly(const ReadOnly &) = delete;
    ReadOnly &operator=(const ReadOnly &) = delete;

  private:
    t val_;
};
```

Motivating example

```
template <typename t>
class ReadOnly {
  public:
    ReadOnly() = default;
    ReadOnly(const t &val) : val_(val) {}
    operator const t &() const { return val_; }
    const t *operator&() const { return &val_; }
    ReadOnly(const ReadOnly &) = delete;
    ReadOnly &operator=(const ReadOnly &) = delete;

  private:
    t val_;
};
```

Works as advertised

```
ReadOnly<int> polarity(23);
polarity = 17; // does not compile
int bar = polarity; // reading works
polarity = 23; // does not compile
```

Advanced implementation techniques

- Looking back at the original/commonly established Design Patterns, *virtually* all of them make use of polymorphism in their implementation

Advanced implementation techniques

- Looking back at the original/commonly established Design Patterns, *virtually* all of them make use of polymorphism in their implementation
- This *runtime polymorphism* works by adding a hidden pointer in every object of a struct/class with virtual functions

Advanced implementation techniques

- Looking back at the original/commonly established Design Patterns, *virtually* all of them make use of polymorphism in their implementation
- This *runtime polymorphism* works by adding a hidden pointer in every object of a struct/class with virtual functions
- This comes with a (huge) runtime cost:

Advanced implementation techniques

- Looking back at the original/commonly established Design Patterns, *virtually* all of them make use of polymorphism in their implementation
- This *runtime polymorphism* works by adding a hidden pointer in every object of a struct/class with virtual functions
- This comes with a (huge) runtime cost:
  - Additional pointer stored per object (8 bytes per object)

Advanced implementation techniques

- Looking back at the original/commonly established Design Patterns, *virtually* all of them make use of polymorphism in their implementation
- This *runtime polymorphism* works by adding a hidden pointer in every object of a struct/class with virtual functions
- This comes with a (huge) runtime cost:
  - Additional pointer stored per object (8 bytes per object)
  - Extra indirection through pointer deference for each invocation of a virtual method

Advanced implementation techniques

- Looking back at the original/commonly established Design Patterns, *virtually* all of them make use of polymorphism in their implementation
- This *runtime polymorphism* works by adding a hidden pointer in every object of a struct/class with virtual functions
- This comes with a (huge) runtime cost:
  - Additional pointer stored per object (8 bytes per object)
  - Extra indirection through pointer deference for each invocation of a virtual method
  - Those virtual methods can hardly be inlined

Advanced implementation techniques

- Admittedly, C++ is hard to understand, write and maintain

Advanced implementation techniques

- Admittedly, C++ is hard to understand, write and maintain
  - Choice for this language is usually due to performance reasons or being used on constrained hardware

Advanced implementation techniques

- Admittedly, C++ is hard to understand, write and maintain
  - Choice for this language is usually due to performance reasons or being used on constrained hardware
  - Hence, these additional costs can be a show stopper, especially in hot code paths

Advanced implementation techniques

- Admittedly, C++ is hard to understand, write and maintain
  - Choice for this language is usually due to performance reasons or being used on constrained hardware
  - Hence, these additional costs can be a show stopper, especially in hot code paths
- In recent years, compilers ship with a *devirtualization* optimization

Advanced implementation techniques

- Admittedly, C++ is hard to understand, write and maintain
  - Choice for this language is usually due to performance reasons or being used on constrained hardware
  - Hence, these additional costs can be a show stopper, especially in hot code paths
- In recent years, compilers ship with a *devirtualization* optimization
  - If an actual type at call site can be determined already at compile-time dynamic dispatch will be resolved statically

Advanced implementation techniques

- Admittedly, C++ is hard to understand, write and maintain
  - Choice for this language is usually due to performance reasons or being used on constrained hardware
  - Hence, these additional costs can be a show stopper, especially in hot code paths
- In recent years, compilers ship with a *devirtualization* optimization
  - If an actual type at call site can be determined already at compile-time dynamic dispatch will be resolved statically
  - In my benchmarks, this only works for the simplest cases (and even then not reliably)

Curiously Recurring Template Pattern (CRTP)

- Combines template (meta) programming with inheritance by

Curiously Recurring Template Pattern (CRTP)

- Combines template (meta) programming with inheritance by
  1. Inheriting from a template class

Curiously Recurring Template Pattern (CRTP)

- Combines template (meta) programming with inheritance by
  1. Inheriting from a template class
  2. Using the derived class itself as a template parameter of the base class

Curiously Recurring Template Pattern (CRTP)

- Combines template (meta) programming with inheritance by
  1. Inheriting from a template class
  2. Using the derived class itself as a template parameter of the base class
- Often referred to as *static polymorphism*, albeit not quite correct as it depends on the use-case

Curiously Recurring Template Pattern (CRTP)

- Combines template (meta) programming with inheritance by
  1. Inheriting from a template class
  2. Using the derived class itself as a template parameter of the base class
- Often referred to as *static polymorphism*, albeit not quite correct as it depends on the use-case
  - Polymorphism provides a single interface or contract for different types

Curiously Recurring Template Pattern (CRTP)

- Combines template (meta) programming with inheritance by
  1. Inheriting from a template class
  2. Using the derived class itself as a template parameter of the base class
- Often referred to as *static polymorphism*, albeit not quite correct as it depends on the use-case
  - Polymorphism provides a single interface or contract for different types
  - CRTP is not related to any interface but a design technique to aid implementation

Curiously Recurring Template Pattern (CRTP)

Motivating Example

```
struct Point {
    explicit Point(int x, int y) : x(x), y(y) {}
    bool operator==(const Point& other) const {
        return x == other.x && y == other.y;
    }
    bool operator!=(const Point& other) const {
        return !(*this == other);
    }

    int x{0}, y{0};
};
```

Curiously Recurring Template Pattern (CRTP)

Motivating Example

```
struct Point {
    explicit Point(int x, int y) : x(x), y(y) {}
    bool operator==(const Point& other) const {
        return x == other.x && y == other.y;
    }
    bool operator!=(const Point& other) const {
        return !(*this == other);
    }

    int x{0}, y{0};
};


struct Rect {
    explicit Rect(int x, int y, int w, int h) : x(x), y(y), w(w), h(h) {}
    bool operator==(const Rect& other) const {
        return x == other.x && y == other.y && w == other.w && h == other.h;
    }
    bool operator!=(const Rect& other) const {
        return !(*this == other);
    }

    int x{0}, y{0}, w{0}, h{0};
};
```

Curiously Recurring Template Pattern (CRTP)

Reducing boilerplate

```cpp
template <typename T>
struct Inequality {
    bool operator!=(const T& other) const {
        const auto& derived = static_cast<const T&>(*this);
        return !(derived == other);
    }
};
```

Curiously Recurring Template Pattern (CRTP)

Reducing boilerplate

```
template <typename T>
struct Inequality {
    bool operator!=(const T& other) const {
        const auto& derived = static_cast<const T&>(*this);
        return !(derived == other);
    }
};


struct Point : public Inequality<Point> {
    explicit Point(int x, int y) : x(x), y(y) {}
    bool operator==(const Point& other) const {
        return x == other.x && y == other.y;
    }

    int x{0}, y{0};
};
```

Curiously Recurring Template Pattern (CRTP)

Reducing boilerplate

```
template <typename T>
struct Inequality {
    bool operator!=(const T& other) const {
        const auto& derived = static_cast<const T&>(*this);
        return !(derived == other);
    }
};


struct Point : public Inequality<Point> {
    explicit Point(int x, int y) : x(x), y(y) {}
    bool operator==(const Point& other) const {
        return x == other.x && y == other.y;
    }

    int x{0}, y{0};
};


struct Rect : public Inequality<Rect> {
    explicit Rect(int x, int y, int w, int h) : x(x), y(y), w(w), h(h) {}
    bool operator==(const Rect& other) const {
        return x == other.x && y == other.y && w == other.w && h == other.h;
    }

    int x{0}, y{0}, w{0}, h{0};
};
```

More flexible CRTP

- With this CRTP we inject *one* feature at compile-time

More flexible CRTP

- With this CRTP we inject *one* feature at compile-time
- Inflexible as we cannot inject further implementation

More flexible CRTP

- With this CRTP we inject *one* feature at compile-time
- Inflexible as we cannot inject further implementation
- Impossible to add additionals to `Point` or `Rect`, e.g. object counter using CRTP

```
template <typename>
struct ObjCounter {
    ObjCounter() { ++size; }
    ObjCounter(const ObjCounter&) { ++size; }
    ~ObjCounter() { --size; }
    static std::size_t size;
};
template <typename T> std::size_t ObjCounter<T>::size = 0;
```

More flexible CRTP

Adding template template parameters

- We can adjust the previous implementation by adding CRTP via template template parameter

More flexible CRTP

Adding template template parameters

- We can adjust the previous implementation by adding CRTP via template template parameter
- Interface for `Point` v2

```
template<template <typename> class Feature>
struct Point : public Feature<Point<Feature>> {};
```

More flexible CRTP

Adding template template parameters

- We can adjust the previous implementation by adding CRTP via template template parameter
- Interface for `Point` v2

```
template<template <typename> class Feature>
struct Point : public Feature<Point<Feature>> {};
```

- With this, `Point` can be specified more freely, e.g.

```
Point<ObjCounter> pt1;
Point<Inequality> pt2;
```

More flexible CRTP

Adding template template parameters

- We can adjust the previous implementation by adding CRTP via template template parameter
- Interface for `Point` v2

```
template<template <typename> class Feature>
struct Point : public Feature<Point<Feature>> {};
```

- With this, `Point` can be specified more freely, e.g.

```
Point<ObjCounter> pt1;
Point<Inequality> pt2;
```

- What about injecting *more than one* features at compile-time?

```
Point<ObjCounter, Inequality, Singular> pt3; // ?
```

Variadic CRTP

- Using variadic template template parameter packs, we can inject several compile-time features at once

Variadic CRTP

- Using variadic template template parameter packs, we can inject several compile-time features at once
- Interface for `Point` v3

```
template<template <typename> class... Feature>
struct Point : public Feature<Point<Feature...>>... {};
```

Variadic CRTP

- Using variadic template template parameter packs, we can inject several compile-time features at once
- Interface for `Point` v3

```
template<template <typename> class... Feature>
struct Point : public Feature<Point<Feature...>>... {};
```

- This works now

```
Point<ObjCounter, Inequality> pt3;
```

Variadic CRTP

Motivating example

Context

Some (embedded) systems only support pull based processing for I/O processing as the processing threads need to be created by ourselves. OTOH most desktop OSes will start their own (realtime) process and provide callbacks.

Variadic CRTP

Motivating example

Context

Some (embedded) systems only support pull based processing for I/O processing as the processing threads need to be created by ourselves. OTOH most desktop OSes will start their own (realtime) process and provide callbacks.

We want to implement something as follows:

```
using MyIODevice = IODevice<PullBasedProcessing>;

MyIODevice device;
device.start();
device.process(io_calback);
device.wait();
...
```

Variadic CRTP

Motivating example (Cont.)

Variadic template template parameter packs to the rescue

```
template <template <typename> class... Feature>
struct IODevice : public Feature<IODevice<Feature...>>... {
  void start() {}
  void stop() {}
  ...
};

using Callback = std::function<void()>;

// Typically on any modern desktop system
template <typename T>
struct PushBasedProcessing {
  void connect(Callback&& io_cb) {}
};

// Typically on embedded, where we need to control everything ourselves
template <typename T>
struct PullBasedProcessing {
  void wait() {}
  void process(Callback&& io_cb) {}
};
```

Summary

- When implemented correctly, CRTP ends up getting recognized by the compiler as static dispatch and optimized away accordingly

Summary

- When implemented correctly, CRTP ends up getting recognized by the compiler as static dispatch and optimized away accordingly
  - Thus, CRTP can be significantly more efficient than dynamic dispatch

Summary

- When implemented correctly, CRTP ends up getting recognized by the compiler as static dispatch and optimized away accordingly
  - Thus, CRTP can be significantly more efficient than dynamic dispatch
  - Note that CRTP can be used to optimize both runtime performance as well as memory footprint

Summary

- When implemented correctly, CRTP ends up getting recognized by the compiler as static dispatch and optimized away accordingly
  - Thus, CRTP can be significantly more efficient than dynamic dispatch
  - Note that CRTP can be used to optimize both runtime performance as well as memory footprint
- Tradeoff between cognitive complexity and performance - productivity gain though

Summary

- When implemented correctly, CRTP ends up getting recognized by the compiler as static dispatch and optimized away accordingly
  - Thus, CRTP can be significantly more efficient than dynamic dispatch
  - Note that CRTP can be used to optimize both runtime performance as well as memory footprint
- Tradeoff between cognitive complexity and performance - productivity gain though
  - May or may not be justifiable depending on team and experience

Tag Dispatching

- Use types to overload on otherwise similar function signatures or constructors

Tag Dispatching

- Use types to overload on otherwise similar function signatures or constructors
- Several ways to call a function or construct object with same types of parameters

Tag Dispatching

- Use types to overload on otherwise similar function signatures or constructors
- Several ways to call a function or construct object with same types of parameters
- A given tag type determines which overload ends up being called

Tag Dispatching

- Use types to overload on otherwise similar function signatures or constructors
- Several ways to call a function or construct object with same types of parameters
- A given tag type determines which overload ends up being called
- Technique used heavily inside STL (e.g. Iterators)

Tag Dispatching

Another motivating example

Context

Create function to calculate profit independent from the actual car vendor or model being used

Tag Dispatching

Another motivating example

Context

Create function to calculate profit independent from the actual car vendor or model being used

```
struct Tesla {
    int getFuelConsumption() const { return 0; }
    int getInsurancePrice() const { return 5; }
    int getFunFactor() const { return 50; }
};
```

Tag Dispatching

Another motivating example

Context

Create function to calculate profit independent from the actual car vendor or model being used

```
struct Tesla {
    int getFuelConsumption() const { return 0; }
    int getInsurancePrice() const { return 5; }
    int getFunFactor() const { return 50; }
};


struct BMW {
    int getBarrelOfFuelPerMeter() const { return 42; }
    int getServiceCosts() const { return 33; }
};
```

Tag Dispatching

Implementation

Final dispatch function

```
template <typename T>
float calcProfit(const T& car) {
    return 17.4f * Profit<T>::output(car) / Profit<T>::input(car);
};
```

Tag Dispatching

Implementation

Final dispatch function

```
template <typename T>
float calcProfit(const T& car) {
    return 17.4f * Profit<T>::output(car) / Profit<T>::input(car);
};
```

Implementation details for each car

```
template <typename T>
struct Profit {};

template <>
struct Profit<Tesla> {
    static int input(const Tesla& t) { return t.getInsurancePrice(); }
    static int output(const Tesla& t) { return t.getFunFactor(); }
};

template <>
struct Profit<BMW> {
    static int input(const BMW& b) { return b.getBarrelOfFuelPerMeter(); }
    static int output(const BMW& b) { return b.getServiceCosts(); }
};
```

Summary

- In most cases, this technique is as good for performance reasons as CRTP

Summary

- In most cases, this technique is as good for performance reasons as CRTP
- Since, this implementation relies on templates, the compiler can see the function definition and optimize accordingly

Summary

- In most cases, this technique is as good for performance reasons as CRTP
- Since, this implementation relies on templates, the compiler can see the function definition and optimize accordingly
- Generally speaking, we are paying with additional compile time for the saved costs in runtime

Outlook

- C++ has strong generic programming capabilities, albeit they can be cumbersome to read **and** write

Outlook

- C++ has strong generic programming capabilities, albeit they can be cumbersome to read **and** write
- The patterns presented in this task only touch the surface

Outlook

- C++ has strong generic programming capabilities, albeit they can be cumbersome to read **and** write
- The patterns presented in this task only touch the surface
- Many more techniques and patterns are actively used out there to be discovered for the interested

Outlook

- C++ has strong generic programming capabilities, albeit they can be cumbersome to read **and** write
- The patterns presented in this task only touch the surface
- Many more techniques and patterns are actively used out there to be discovered for the interested
  - Type Erasure
  - Type Traits

Outlook

- C++ has strong generic programming capabilities, albeit they can be cumbersome to read **and** write
- The patterns presented in this task only touch the surface
- Many more techniques and patterns are actively used out there to be discovered for the interested
  - Type Erasure
  - Type Traits
  - Expression Templates

Outlook

- C++ has strong generic programming capabilities, albeit they can be cumbersome to read **and** write
- The patterns presented in this task only touch the surface
- Many more techniques and patterns are actively used out there to be discovered for the interested
  - Type Erasure
  - Type Traits
  - Expression Templates

Outlook

- C++ has strong generic programming capabilities, albeit they can be cumbersome to read **and** write
- The patterns presented in this task only touch the surface
- Many more techniques and patterns are actively used out there to be discovered for the interested
  - Type Erasure
  - Type Traits
  - Expression Templates
  - ...

Commercial Break

Available for consulting services, trainings and workshops. Feel free to contact me at www.kai-wolf.me

Thank you