Projektowanie złożonych systemów telekomunikacyjnych

Templates & Concepts

Aleksander Miera

Agenda

1. Introduction
2. Class templates (and type aliases)
3. Variable templates
4. Function templates
5. Forwarding references
6. Concepts and constraints
7. Advanced topics, caveats, pitfalls
8. Learning material

Introduction

Introduction

Templates

- Feature of C++ that enables generic programming
- Feature of C++ that enables compile-time metaprogramming
  - Initially purely functional style
  - ...more imperative compile-time programming with introduction of constexpr

Class templates

Basic syntax

Type parameter

Non-type parameter (can be auto)

```
template<typename T, std::size_t S>
struct Array
{
    T contents[S];
    //...
}

int main(int, char**)
{
  Array<int, 3u> ai{1,2,3};
  Array<double, 8> ad{};
}
```

Note the types are unrelated

Generally manually specified, can be deduced from class's constructor since C++17, expanded in C++20. Use with caution, caveats may apply!

Templates are inline by default

https://godbolt.org/z/KhPhTnvPr

Class templates

Basic syntax

Multiple type parameters

```
template<typename T1, typename T2>
struct pair
{
  T1 first;
  T2 second;
};
```

std::pair

Defined in header <utility>

```
template<
    class T1,
    class T2
> struct pair;
```

https://en.cppreference.com/w/cpp/utility/pair

Class templates

Basic syntax

Template parameters can have default values

Rules similar to function parameters – defaults from right to left

std::**unique_ptr**

Defined in header <memory>

```
template<
    class T,
    class Deleter = std::default_delete<T>
> class unique_ptr;
```

(1) (since C++11)

```
template <
    class T,
    class Deleter
> class unique_ptr<T[], Deleter>;
```

(2) (since C++11)

https://en.cppreference.com/w/cpp/memory/unique_ptr

std::**vector**

Defined in header <vector>

```
template<
    class T,
    class Allocator = std::allocator<T>
> class vector;
```

https://en.cppreference.com/w/cpp/container/vector

Class templates

Basic syntax

Class templates can be variadic, i.e. with various number of parameters

std::tuple

Defined in header <tuple>

```
template< class... Types >
class tuple;
```

(since C++11)

https://en.cppreference.com/w/cpp/utility/tuple

Class templates

Basic syntax

Class templates can be variadic, i.e. with various number of parameters

This is called parameter pack.

Generally (there are some exceptions), parametr packs should appear as the last template parameter.

std::tuple

Defined in header <tuple>

```
template< class... Types >
class tuple;
```

(since C++11)

https://en.cppreference.com/w/cpp/utility/tuple

Class templates
Basic syntax

```
#include <iostream>

template<typename ...Functor>
struct OverloadedFunctor : Functor...
{
  using Functor::operator()...;
};

struct IntPrinter
{
  void operator()(int x) const { std::cout << x << '\n';}
};

struct DoublePrinter
{
  void operator()(double x) const { std::cout << x << '\n';}
};

int main(int, char**)
{
    OverloadedFunctor<IntPrinter, DoublePrinter> p;
    p(3);
    p(3.14);
}
```

Type’s name in parameter pack refers to its first element, the rest is expanded using ...

https://godbolt.org/z/nPa3d4sfY

Class templates

Basic syntax

```
#include <iostream>

template<typename Functor, typename ...Rest>
struct OverloadedFunctor : Functor, OveloadedFunctor<Rest...>
{
  using Functor::operator()…;
  using OverloadedFunctor<Rest…>::operator();
};

template<typename Functor>
struct OverloadedFunctor<Functor> : Functor
{
  using Functor::operator()...;
};

struct IntPrinter
{
  void operator()(int x) const { std::cout << x << '\n';}
};

struct DoublePrinter
{
  void operator()(double x) const { std::cout << x << '\n';}
};

int main(int, char**)
{
    OverloadedFunctor<IntPrinter, DoublePrinter> p;
    p(3);
    p(3.14);
}
```

Partial specialization

Type’s name in parameter pack refers to its first element, the rest is expanded using ...

or recursively (due to lack of some features in standards predating C++17)

https://godbolt.org/z/daejch7TP

Class templates

Basic syntax

Type's name in parameter pack refers to its first element, the rest is expanded using ...

or recursively (due to lack of some features in standards predating C++17)

Number of elements is return by **sizeof...(args)**

```
#include <iostream>

template<typename Functor, typename ...Rest>
struct OverloadedFunctor : Functor, OveloadedFunctor<Rest...>
{
  using Functor::operator()…;
  using OverloadedFunctor<Rest…>::operator();
};

template<typename Functor>
struct OverloadedFunctor<Functor> : Functor
{
  using Functor::operator()...;
};

struct IntPrinter
{
  void operator()(int x) const { std::cout << x << '\n';}
};

struct DoublePrinter
{
  void operator()(double x) const { std::cout << x << '\n';}
};

int main(int, char**)
{
    OverloadedFunctor<IntPrinter, DoublePrinter> p;
    p(3);
    p(3.14);
    return OverloadedFunctor<IntPrinter, DoublePrinter>::numberOfParams;
}
```

Partial specialization

https://godbolt.org/z/GTYGxnbqf

Class templates

Basic syntax

Specializations allow specific behaviour of a template, e.g.:

1. Using functions specific for a type parameter

```
template<typename Container>
class AppendingContainerWrapper
{
public:
    explicit AppendingContainerWrapper(Container& c)
    : container(std::addressof(c)) {}
    void push_back(const typename Container::value_type& val)
    {
        std::cout << "other\n";
        container->insert(container->end(), val);
    }
private:
    Container* container;
};

template<typename T>
class AppendingContainerWrapper<std::vector<T>>
{
public:
    explicit AppendingContainerWrapper(std::vector<T>& v)
    : container(std::addressof(v)) {}
    void push_back(const T& val)
    {
        std::cout << "vec\n";
        container->push_back(val);
    }
private:
    std::vector<T>* container;
};
```

typename keyword for dependent types

https://godbolt.org/z/T37cvrosY

Class templates

Basic syntax

Specializations allow specific behaviour of a template, e.g.:

1. Using functions specific for a type parameter
2. Identify type (type traits)

```
#include <type_traits>

template<typename>
struct isPointer : std::false_type{};

template<typename T>
struct isPointer<T*> : std::true_type{};

template<typename T>
struct isPointer<T* const> : std::true_type{};

template<typename T>
struct isPointer<T* volatile> : std::true_type{};

template<typename T>
struct isPointer<T* const volatile> : std::true_type{};



static_assert(isPointer<int*>::value, "");
static_assert(isPointer<const double* volatile>::value, "");
static_assert(isPointer<const float* const>::value, "");
static_assert(!isPointer<int>::value, "");
```

https://godbolt.org/z/fx9Eo836P

Class templates

Basic syntax

Similar for full specializations

```
#include <type_traits>

template<typename>
struct isBool : std::false_type{};

template<>
struct isBool<bool> : std::true_type{};

static_assert(!isBool<int*>::value, "");
static_assert(!isBool<char>::value, "");
static_assert(isBool<bool>::value, "");
```

https://godbolt.org/z/YYMPrEeGo

Class templates

Basic syntax

Similar for full specializations

…even custom member functions can be added, see

std::vector<bool>

Modifiers

`clear`	clears the contents (public member function of std::vector<T,Allocator>)
`insert`	inserts elements (public member function of std::vector<T,Allocator>)
`insert_range` (C++23)	inserts a range of elements (public member function of std::vector<T,Allocator>)
`append_range` (C++23)	adds a range of elements to the end (public member function of std::vector<T,Allocator>)
`emplace` (C++11)	constructs element in-place (public member function of std::vector<T,Allocator>)
`erase`	erases elements (public member function of std::vector<T,Allocator>)
`push_back`	adds an element to the end (public member function of std::vector<T,Allocator>)
`emplace_back` (C++11)	constructs an element in-place at the end (public member function of std::vector<T,Allocator>)
`pop_back`	removes the last element (public member function of std::vector<T,Allocator>)
`resize`	changes the number of elements stored (public member function of std::vector<T,Allocator>)
`swap`	swaps the contents (public member function of std::vector<T,Allocator>)

vector<bool> specific modifiers

`flip`	flips all the bits (public member function)
`swap` [static]	swaps two std::vector<bool>::references (public static member function)

https://en.cppreference.com/w/cpp/container/vector_bool

Variable templates

Basic syntax

Can be used as type traits

```
#include <type_traits>


template<typename>
constexpr bool isBool = false;

template<>
constexpr bool isBool<bool> = true;


static_assert(!isBool<int*>, "");
static_assert(!isBool<char>, "");
static_assert(isBool<bool>, "");
```

https://godbolt.org/z/r7MGxdfj3

Variable templates

Basic syntax

1. Can be used as type traits
2. Or as named parameters

```
void processParamAccordingToFeature(bool featureOn, int paramVal);

template<bool VAL>
constexpr bool featureFlagOn = VAL;

template<int VAL>
constexpr int importantParam = VAL;

int main(int, char**)
{
    processParamAccordingToFeature(featureFlagOn<false>, importantParam<32>);
    processParamAccordingToFeature(featureFlagOn<true>, importantParam<92>);
}
```

https://godbolt.org/z/6Ev8zE7M5

Function templates

Basic syntax

- Type parameter (can be default)
- It can be deduced at compile time from here

```
template<typename S>
auto get_listening_socket(S& x)
{
    return x.listen();
}
```

Note the types are unrelated

```
struct UdpSocket { UdpSocket listen() { return {}; } };
struct TcpSocket { TcpSocket listen() { return {}; } };
int main(int, char**)
{
    UdpSocket u;
    TcpSocket t;
    auto ul = get_listening_socket(u);
    auto tl = get_listening_socket(t);
}
```

https://godbolt.org/z/orbern9Gs

Function templates

Basic syntax

Non template type parameter

Template type parameter pack

```
template<int X, typename...Args>
auto allEqualTo(Args&&... args)
{
    return ((X == args)&&...);
}

int main(int, char**)
{
  std::cout << allEqual<3>(1,3,5,8) << '\n';
  std::cout << allEqual<22>(22,22,22,22) << '\n';
}
```

Types in the pack deduced at compile time

Fold expression: parameter pack being fold over an expression

https://godbolt.org/z/dbcE8eTPf

```
bool allEqual<3, int, int, int, int>(int&&, int&&, int&&, int&&):
  pushq %rbp
  movq %rsp, %rbp
  movq %rdi, -8(%rbp)
  movq %rsi, -16(%rbp)
  movq %rdx, -24(%rbp)
  movq %rcx, -32(%rbp)
  movq -8(%rbp), %rax
  movl (%rax), %eax
  cmpl $3, %eax
  jne .L4
  movq -16(%rbp), %rax
  movl (%rax), %eax
  cmpl $3, %eax
  jne .L4
  movq -24(%rbp), %rax
  movl (%rax), %eax
  cmpl $3, %eax
  jne .L4
  movq -32(%rbp), %rax
  movl (%rax), %eax
  cmpl $3, %eax
  jne .L4
  movl $1, %eax
  jmp .L5
.L4:
  movl $0, %eax
.L5:
  popq %rbp
  ret
bool allEqual<22, int, int, int>(int&&, int&&, int&&):
```

Function templates

Basic syntax

```
template<typename T>
void clearVec(std::vector<T>& x)
{
    x.clear();
}

template<typename T, typename U>
T expandPrecision(U x)
{
    static_assert(sizeof(T) >= sizeof(U));
    return static_cast<T>(x);
}

int main(int, char**)
{
    std::vector<int> vi{1,2,3};
    short s = 32;
    long l = expandPrecision<long>(s);
}
```

Type of x is **not considered deduced**

T cannot be deduced

U is deduced

https://godbolt.org/z/rPc9Y8hMP

Function templates

Basic syntax

```
template<typename T>
void clearVec(std::vector<T>& x)
{
    x.clear();
}

template<typename T, typename U>
T expandPrecision(U x)
{
    static_assert(sizeof(T) >= sizeof(U));
    return static_cast<T>(x);
}

int main(int, char**)
{
    std::vector<int> vi{1,2,3};
    short s = 32;
    long l = expandPrecision<long, const int&>(s);
}
```

Template type arameter can be specified manually to avoid deduction – regular caveats related to conversion apply – **GENERALLY DISCOURAGED WHEN DEDUCTION IS AVAILABLE**

https://godbolt.org/z/qvEqqrqzK

Function templates

Are they really functions?

```
template<typename S>
auto get_listening_socket(S& x)
{
    return x.listen();
}

struct UdpSocket { UdpSocket listen() { return {}; } };
struct TcpSocket { TcpSocket listen() { return {}; } };
int main(int, char**)
{
    UdpSocket u;
    TcpSocket t;
    //auto ul = get_listening_socket(u);
    //auto tl = get_listening_socket(t);
}
```

```
main:
  pushq %rbp
  movq %rsp, %rbp
  movl %edi, -20(%rbp)
  movq %rsi, -32(%rbp)
  movl $0, %eax
  popq %rbp
  ret
```

https://godbolt.org/z/TEGcP4s7W

Function templates

Are they really functions?

```
template<typename S>
auto get_listening_socket(S& x)
{
    return x.listen();
}

struct UdpSocket { UdpSocket listen() { return {}; } }
struct TcpSocket { TcpSocket listen() { return {}; } }
//expclit instantiations:
template auto get_listening_socket(UdpSocket&);
template auto get_listening_socket(TcpSocket&);

int main(int, char**)
{
    UdpSocket u;
    TcpSocket t;
}
```

```
auto get_listening_socket<UdpSocket>(UdpSocket&):
  pushq %rbp
  movq %rsp, %rbp
  subq $16, %rsp
  movq %rdi, -8(%rbp)
  movq -8(%rbp), %rax
  movq %rax, %rdi
  call UdpSocket::listen()
  nop
  leave
  ret
auto get_listening_socket<TcpSocket>(TcpSocket&):
  pushq %rbp
  movq %rsp, %rbp
  subq $16, %rsp
  movq %rdi, -8(%rbp)
  movq -8(%rbp), %rax
  movq %rax, %rdi
  call TcpSocket::listen()
  nop
  leave
  ret
```

https://godbolt.org/z/PxcqEo1TY

Function templates

Are they really functions?

```
template<typename S>
auto get_listening_socket(S& x)
{
    return x.listen();
}

struct UdpSocket { UdpSocket listen() { return {}; } };
struct TcpSocket { TcpSocket listen() { return {}; } };
int main(int, char**)
{
    UdpSocket u;
    TcpSocket t;
    auto ul = get_listening_socket(u);
    auto tl = get_listening_socket(t);
}
```

https://godbolt.org/z/zY3a3E6dv

A ▾ Output... ▾ Filter... ▾ Libraries Overrides + Add new... ▾

```
auto get_listening_socket<UdpSocket>(UdpSocket&):
  pushq %rbp
  movq %rsp, %rbp
  subq $16, %rsp
  movq %rdi, -8(%rbp)
  movq -8(%rbp), %rax
  movq %rax, %rdi
  call UdpSocket::listen()
  nop
  leave
  ret
auto get_listening_socket<TcpSocket>(TcpSocket&):
  pushq %rbp
  movq %rsp, %rbp
  subq $16, %rsp
  movq %rdi, -8(%rbp)
  movq -8(%rbp), %rax
  movq %rax, %rdi
  call TcpSocket::listen()
  nop
  leave
  ret
main:
  pushq %rbp
  movq %rsp, %rbp
  subq $32, %rsp
  movl %edi, -20(%rbp)
  movq %rsi, -32(%rbp)
  leaq -1(%rbp), %rax
  movq %rax, %rdi
  call auto get_listening_socket<UdpSocket>(UdpSocket&)
  leaq -2(%rbp), %rax
  movq %rax, %rdi
  call auto get_listening_socket<TcpSocket>(TcpSocket&)
  movl $0, %eax
  leave
  ret
```


Function templates

What about class template member functions?

```
template<typename T>
struct Temperature
{
    T value;
    explicit Temperature(T x) : value(x) {}
    Temperature lowerBy(T x) const { return Temperature{value-x};}
    Temperature higherBy(T x) const { return value+x;}
};
// template struct Temperature<int>;

int main(int, char**)
{
    Temperature<int> t{3};
    t.lowerBy(8);
}
```

```
main:
  pushq %rbp
  movq %rsp, %rbp
  subq $32, %rsp
  movl %edi, -20(%rbp)
  movq %rsi, -32(%rbp)
  leaq -4(%rbp), %rax
  movl $3, %esi
  movq %rax, %rdi
  call Temperature<int>::Temperature(int) [complete object constructor]
  leaq -4(%rbp), %rax
  movl $8, %esi
  movq %rax, %rdi
  call Temperature<int>::lowerBy(int) const
  movl $0, %eax
  leave
  ret
Temperature<int>::Temperature(int) [base object constructor]:
  pushq %rbp
  movq %rsp, %rbp
  movq %rdi, -8(%rbp)
  movl %esi, -12(%rbp)
  movq -8(%rbp), %rax
  movl -12(%rbp), %edx
  movl %edx, (%rax)
  nop
  popq %rbp
  ret
Temperature<int>::lowerBy(int) const:
  pushq %rbp
  movq %rsp, %rbp
  subq $32, %rsp
  movq %rdi, -24(%rbp)
  movl %esi, -28(%rbp)
  movq -24(%rbp), %rax
  movl (%rax), %eax
  subl -28(%rbp), %eax
  movl %eax, %edx
  leaq -4(%rbp), %rax
  movl %edx, %esi
  movq %rax, %rdi
  call Temperature<int>::Temperature(int) [complete object constructor]
  movl -4(%rbp), %eax
  leave
  ret
```

https://godbolt.org/z/s76oe7sEW

Function templates

What about class template member functions?

```cpp
template<typename T>
struct Temperature
{
    T value;
    explicit Temperature(T x) : value(x) {}
    Temperature lowerBy(T x) const { return Temperature{value-x};}
    Temperature higherBy(T x) const { return value+x;}
};
template struct Temperature<int>;

int main(int, char**)
{
    Temperature<int> t{3};
    t.lowerBy(8);
}
```

Can you spot the problem?

https://godbolt.org/z/TbnP3Y63s

```asm
main:
  pushq %rbp
  movq %rsp, %rbp
  subq $32, %rsp
  movl %edi, -20(%rbp)
  movq %rsi, -32(%rbp)
  leaq -4(%rbp), %rax
  movl $3, %esi
  movq %rax, %rdi
  call Temperature<int>::Temperature(int) [complete object constructor]
  leaq -4(%rbp), %rax
  movl $8, %esi
  movq %rax, %rdi
  call Temperature<int>::lowerBy(int) const
  movl $0, %eax
  leave
  ret
Temperature<int>::Temperature(int) [base object constructor]:
  pushq %rbp
  movq %rsp, %rbp
  movq %rdi, -8(%rbp)
  movl %esi, -12(%rbp)
  movq -8(%rbp), %rax
  movl -12(%rbp), %edx
  movl %edx, (%rax)
  nop
  popq %rbp
  ret
Temperature<int>::lowerBy(int) const:
  pushq %rbp
  movq %rsp, %rbp
  subq $32, %rsp
  movq %rdi, -24(%rbp)
  movl %esi, -28(%rbp)
  movq -24(%rbp), %rax
  movl (%rax), %eax
  subl -28(%rbp), %eax
  movl %eax, %edx
  leaq -4(%rbp), %rax
  movl %edx, %esi
  movq %rax, %rdi
  call Temperature<int>::Temperature(int) [complete object constructor]
  movl -4(%rbp), %eax
  leave
  ret
```


Function templates

What about class template member functions?

```
template<typename T>
struct Temperature
{
    T value;
    explicit Temperature(T x) : value(x) {}
    Temperature lowerBy(T x) const { return Temperature{value-x};}
    Temperature higherBy(T x) const { return value+x;}
};
template struct Temperature<int>;

int main(int, char**)
{
    Temperature<int> t{3};
    t.lowerBy(8);
}
```

Those are not function templates!

https://godbolt.org/z/TbnP3Y63s

Function templates

What about class template member functions?

```
template<typename T>
struct Temperature
{
    T value;
    explicit Temperature(T x) : value(x) {}
    Temperature lowerBy(T x) const { return Temperature{value-x};}
    Temperature higherBy(T x) const { return Temperature{value+x};}
};

template struct Temperature<int>;
int main(int, char**)
{
    Temperature<int> t{3};
    t.lowerBy(8);
}
```

```
Temperature<int>::lowerBy(int) const:
  pushq %rbp
  movq %rsp, %rbp
  subq $32, %rsp
  movq %rdi, -24(%rbp)
  movl %esi, -28(%rbp)
  movq -24(%rbp), %rax
  movl (%rax), %eax
  subl -28(%rbp), %eax
  movl %eax, %edx
  leaq -4(%rbp), %rax
  movl %edx, %esi
  movq %rax, %rdi
  call Temperature<int>::Temperature(int) [complete object constructor]
  movl -4(%rbp), %eax
  leave
  ret
Temperature<int>::higherBy(int) const:
  pushq %rbp
  movq %rsp, %rbp
  subq $32, %rsp
  movq %rdi, -24(%rbp)
  movl %esi, -28(%rbp)
  movq -24(%rbp), %rax
  movl (%rax), %edx
  movl -28(%rbp), %eax
  addl %eax, %edx
  leaq -4(%rbp), %rax
  movl %edx, %esi
  movq %rax, %rdi
  call Temperature<int>::Temperature(int) [complete object constructor]
  movl -4(%rbp), %eax
  leave
  ret
```

https://godbolt.org/z/dq1avf4vz

Function templates

What about class template member functions?

```
template<typename T>
struct Temperature
{
    T value;
    explicit Temperature(T x) : value(x) {}
    template<std::same_as<T> U>
    Temperature lowerBy(U x) const { return Temperature{value-x};}

    template<std::same_as<T> U>
    Temperature higherBy(U x) const { return Temperature{value+x};}
};

template struct Temperature<int>;
int main(int, char**)
{
    Temperature<int> t{3};
    t.lowerBy(8);
}
```

https://godbolt.org/z/hrvs78T57

```
Temperature<int>::lowerBy(int) const:
  pushq %rbp
  movq %rsp, %rbp
  subq $32, %rsp
  movq %rdi, -24(%rbp)
  movl %esi, -28(%rbp)
  movq -24(%rbp), %rax
  movl (%rax), %eax
  subl -28(%rbp), %eax
  movl %eax, %edx
  leaq -4(%rbp), %rax
  movl %edx, %esi
  movq %rax, %rdi
  call Temperature<int>::Temperature(int) [complete object constructor]
  movl -4(%rbp), %eax
  leave
  ret
Temperature<int>::higherBy(int) const:
  pushq %rbp
  movq %rsp, %rbp
  subq $32, %rsp
  movq %rdi, -24(%rbp)
  movl %esi, -28(%rbp)
  movq -24(%rbp), %rax
  movl (%rax), %edx
  movl -28(%rbp), %eax
  addl %eax, %edx
  leaq -4(%rbp), %rax
  movl %edx, %esi
  movq %rax, %rdi
  call Temperature<int>::Temperature(int) [complete object constructor]
  movl -4(%rbp), %eax
  leave
  ret
```


Function templates

Compared to overloads

```cpp
struct UdpSocket { UdpSocket listen() { return {}; } };
struct TcpSocket { TcpSocket listen() { return {}; } };

auto get_listening_socket(UdpSocket& x)
{
    return x.listen();
}

auto get_listening_socket(TcpSocket& x)
{
    return x.listen();
}

int main(int, char**)
{
    UdpSocket u;
    TcpSocket t;
    // auto ul = get_listening_socket(u);
    // auto tl = get_listening_socket(t);
}
```

```asm
  pushq %rbp
  movq %rsp, %rbp
  movq %rdi, -8(%rbp)
  nop
  popq %rbp
  ret
get_listening_socket(UdpSocket&):
  pushq %rbp
  movq %rsp, %rbp
  subq $16, %rsp
  movq %rdi, -8(%rbp)
  movq -8(%rbp), %rax
  movq %rax, %rdi
  call UdpSocket::listen()
  nop
  leave
  ret
get_listening_socket(TcpSocket&):
  pushq %rbp
  movq %rsp, %rbp
  subq $16, %rsp
  movq %rdi, -8(%rbp)
  movq -8(%rbp), %rax
  movq %rax, %rdi
  call TcpSocket::listen()
  nop
  leave
  ret
main:
  pushq %rbp
  movq %rsp, %rbp
  movl %edi, -20(%rbp)
  movq %rsi, -32(%rbp)
  movl $0, %eax
  popq %rbp
  ret
```

https://godbolt.org/z/ds1M4xvqv

Function templates

Compared to full specializations

```
#include <cstdint>

struct UdpSocket {
    void send(char*, std::uint32_t);
};
struct TcpSocket {
    void tryToConnectAndSend(char*, std::uint32_t);
};
template<typename SOCKET>
void sendData(SOCKET& s, char* buf, std::uint32_t addr)
{
    return s.send(buf, addr);
}
template<>
//inline
void sendData<TcpSocket>(TcpSocket& s, char* buf, std::uint32_t addr)
{
    return s.tryToConnectAndSend(buf, addr);
}
int main(int, char**)
{
    UdpSocket u;
    TcpSocket t;
    // sendData(u, nullptr, 0);
    // sendData(t, nullptr, 0);
}
```

Uncomment and see what happens
Notice linker error!

```
void sendData<TcpSocket>(TcpSocket&, char*, unsigned int):
  pushq %rbp
  movq %rsp, %rbp
  subq $32, %rsp
  movq %rdi, -8(%rbp)
  movq %rsi, -16(%rbp)
  movl %edx, -20(%rbp)
  movl -20(%rbp), %edx
  movq -16(%rbp), %rcx
  movq -8(%rbp), %rax
  movq %rcx, %rsi
  movq %rax, %rdi
  call TcpSocket::tryToConnectAndSend(char*, unsigned int)
  nop
  leave
  ret
main:
  pushq %rbp
  movq %rsp, %rbp
  movl %edi, -20(%rbp)
  movq %rsi, -32(%rbp)
  movl $0, %eax
  popq %rbp
  ret
```

https://godbolt.org/z/ccd3KqMWW

Function templates

Compared to full specializations

```
#include <cstdint>

struct UdpSocket {
    void send(char*, std::uint32_t);
};
struct TcpSocket {
    void tryToConnectAndSend(char*, std::uint32_t);
};
template<typename SOCKET>
void sendData(SOCKET& s, char* buf, std::uint32_t addr)
{
    return s.send(buf, addr);
}
template<>
//inline
void sendData<TcpSocket>(TcpSocket& s, char* buf, std::uint32_t addr)
{
    return s.tryToConnectAndSend(buf, addr);
}
int main(int, char**)
{
    UdpSocket u;
    TcpSocket t;
    // sendData(u, nullptr, 0);
    // sendData(t, nullptr, 0);
}
```

NO PARTIAL SPECIALIZATIONS FOR FUNCTION TEMPLATES

Uncomment and see what happens
Notice linker error!

https://godbolt.org/z/ccd3KqMWW

Function templates

Argument deduction, conversions, constraining argument types

```
void doStuff(bool);

int main(int, char**)
{
    doStuff(true);
    int x = 3;
    doStuff(&x); //OOPS
}
```

https://godbolt.org/z/4h9T3Y1q8

Function templates

Argument deduction, conversions, constraining argument types

```
template<typename T>
void doStuff(T)
{
//whatever here
}

int main(int, char**)
{
    doStuff(true); //OK
    int x = 3;
    doStuff(&x); //better, separate function but still compiles
}
```

```
void doStuff<bool>(bool):
  pushq %rbp
  movq %rsp, %rbp
  movl %edi, %eax
  movb %al, -4(%rbp)
  nop
  popq %rbp
  ret
void doStuff<int*>(int*):
  pushq %rbp
  movq %rsp, %rbp
  movq %rdi, -8(%rbp)
  nop
  popq %rbp
  ret
```

https://godbolt.org/z/94cKcGoxK

Function templates

Argument deduction, conversions, constraining argument types

```
template<typename T>
requires std::same_as<T, bool> // C++20
void doStuff(T)
{
}

int main(int, char**)
{
    doStuff(true); //OK
    int x = 3;
    doStuff(&x); //OK, compile error!
}
```

https://godbolt.org/z/xEsnfx8T8

Function templates

Argument deduction, conversions, constraining argument types

```
template<typename T>
requires std::same_as<T, bool> // C++20
void doStuff(T)
{
}

int main(int, char**)
{
    doStuff(true); //OK
    int x = 3;
    doStuff(&x); //OK, compile error!
}
```

https://godbolt.org/z/xEsnfx8T8

```
template<typename T>
void doStuff(T) = delete;
void doStuff(bool);
// pre C++20, not entirely equivalent

int main(int, char**)
{
    doStuff(true); //OK
    int x = 3;
    doStuff(&x); //OK, compile error!
}
```

https://godbolt.org/z/6n4zjKnTj

Function templates

Argument deduction, conversions, constraining argument types

```
#include<concepts>
template<typename T>
requires std::same_as<T, bool> // C++20
void doStuff(T)
{
}

int main(int, char**)
{
    doStuff(true); //OK
    int x = 3;
    doStuff(&x); //OK, compile error!
}
```

https://godbolt.org/z/xEsnfx8T8

```
template<typename T>
void doStuff(T) = delete;
void doStuff(bool);
// pre C++20, not entirely equivalent

int main(int, char**)
{
    doStuff(true); //OK
    int x = 3;
    doStuff(&x); //OK, compile error!
}
```

https://godbolt.org/z/6n4zjKnTj

Function templates

Argument deduction, conversions, constraining argument types

```
#include <type_traits>
template<typename T>
void doStuff(T)
{
   static_assert(std::is_same_v<T, bool>, "!");
}

int main(int, char**)
{
    doStuff(true); //OK
    int x = 3;
    doStuff(&x); //OK, compile error
                 //but a bit late
}
```

https://godbolt.org/z/PMWvbz7fP

Function templates

Argument deduction, conversions, constraining argument types

```
#include <type_traits>
template<typename T>
void doStuff(T)
{
   static_assert(std::is_same_v<T, bool>, "!");
}

int main(int, char**)
{
    doStuff(true); //OK
    int x = 3;
    doStuff(&x); //OK, compile error
                 //but a bit late
}
```

```
#include <type_traits>
template<typename T, std::enable_if_t<std::is_same_v<T, bool>>*=nullptr>
void doStuff(T)
{

}

int main(int, char**)
{
    doStuff(true); //OK
    int x = 3;
    doStuff(&x); //OK, compile error
}
```

https://godbolt.org/z/PMWvbz7fP https://godbolt.org/z/Es1a7Maq9

Function templates

Argument deduction, conversions, constraining argument types

```
#include <type_traits>
template<typename T>
void doStuff(T)
{
   static_assert(std::is_same_v<T, bool>, "!");
}

int main(int, char**)
{
    doStuff(true); //OK
    int x = 3;
    doStuff(&x); //OK, compile error
                 //but a bit late
}
```

```
#include <type_traits>
template<typename T, std::enable_if_t<std::is_same_v<T, bool>>*=nullptr>
void doStuff(T)
{

}

int main(int, char**)
{
    doStuff(true); //OK
    int x = 3;
    doStuff(&x); //OK, compile error
}
```

This works, but is ugly

https://godbolt.org/z/PMWvbz7fP https://godbolt.org/z/Es1a7Maq9

Function templates

Argument deduction, conversions, constraining argument types

```
#include <type_traits>
template<typename T>
void doStuff(T)
{
   static_assert(std::is_same_v<T, bool>, "!");
}

int main(int, char**)
{
    doStuff(true); //OK
    int x = 3;
    doStuff(&x); //OK, compile error
                 //but a bit late
}
```

```
#include <type_traits>
template<typename T, std::enable_if_t<std::is_same_v<T, bool>>* = nullptr>
void doStuff(T)
{

}

int main(int, char**)
{
    doStuff(true); //OK
    int x = 3;
    doStuff(&x); //OK, compile error
}
```

This works, but is ugly
Yes, SFINAE can be ugly ☹

https://godbolt.org/z/PMWvbz7fP https://godbolt.org/z/Es1a7Maq9

Function templates

Argument deduction, conversions, constraining argument types

```
#include <type_traits>
template<typename T>
void doStuff(T)
{
   static_assert(std::is_same_v<T, bool>, "!");
}

int main(int, char**)
{
    doStuff(true); //OK
    int x = 3;
    doStuff(&x); //OK, compile error
                 //but a bit late
}
```

```
#include <type_traits>
template<typename T, std::enable_if_t<std::is_same_v<T, bool>>* = nullptr>
void doStuff(T)
{

}

int main(int, char**)
{
    doStuff(true); //OK
    int x = 3;
    doStuff(&x); //OK, compile error
}
```

This works, but is ugly
Yes, SFINAE can be ugly ☹
SFINAE? What is that?

https://godbolt.org/z/PMWvbz7fP https://godbolt.org/z/Es1a7Maq9

Function templates

SFINAE – Substitution Failure Is Not An Error

Consider the previous example

```
#include <type_traits>
template<typename T, std::enable_if_t<std::is_same_v<T, bool>>* = nullptr>
void doStuff(T)
{

}

int main(int, char**)
{
    doStuff(true); //OK
    int x = 3;
    doStuff(&x); //how to make it work for pointers?
}
```

https://godbolt.org/z/PMWvbz7fP https://godbolt.org/z/Es1a7Maq9

Function templates

SFINAE – Substitution Failure Is Not An Error

Consider the previous example

```
#include <type_traits>
template<typename T, std::enable_if_t<std::is_same_v<T, bool>>* = nullptr>
void doStuff(T)
{

}

template<typename T>
void doStuff(T)
{

}

int main(int, char**)
{
    doStuff(true); //OK
    int x = 3;
    doStuff(&x); //how to make it work for pointers?
}
```


Function templates

SFINAE – Substitution Failure Is Not An Error

Consider the previous example

```
#include <type_traits>
template<typename T, std::enable_if_t<std::is_same_v<T, bool>>* = nullptr>
void doStuff(T)
{

}

template<typename T>
void doStuff(T)  //nope, call is ambiguous
{

}

int main(int, char**)
{
    doStuff(true); //OK
    int x = 3;
    doStuff(&x); //how to make it work for pointers?
}
```

https://godbolt.org/z/93fxfMzGf

Function templates

SFINAE – Substitution Failure Is Not An Error

Consider the previous example

```
#include <type_traits>
template<typename T, std::enable_if_t<std::is_same_v<T, bool>>* = nullptr>
void doStuff(T)
{

}

template<typename T>
void doStuff(T)  //nope, call is ambiguous
{

}

int main(int, char**)
{
    doStuff(true); //OK
    int x = 3;
    doStuff(&x); //how to make it work for pointers?
}
```

https://godbolt.org/z/6qvbe5KqE

Function templates

SFINAE – Substitution Failure Is Not An Error

Consider the previous example

```
#include <type_traits>
template<typename T, std::enable_if_t<std::is_same_v<T, bool>>* = nullptr>
void doStuff(T)
{

}

template<typename T>
auto doStuff(T) -> std::enable_if_t<std::is_pointer_v<T>>
{

}

int main(int, char**)
{
    doStuff(true); //OK
    int x = 3;
    doStuff(&x); //now it works
}
```

https://godbolt.org/z/cG948qoj3

Function templates

SFINAE – Substitution Failure Is Not An Error

Now make it contemporary

```
#include <concepts>
#include <type_traits>
template<std::same_as<bool> T>
//requires std::same_as<bool, T>
//requires std::is_same_v<bool, T> && std::is_same_v<T, bool>
void doStuff(T)
{
}

template<typename T>
requires std::is_pointer_v<T>
void doStuff(T)
{
}

int main(int, char**)
{
    doStuff(true); //OK
    int x = 3;
    doStuff(&x); //now it works
}
```

https://godbolt.org/z/Kd1oo9q3G

Function templates

SFINAE – Substitution Failure Is Not An Error

Or using some other notation

https://godbolt.org/z/frTrEv4W3

```
#include <concepts>
#include <type_traits>

template<typename T>
constexpr bool dependent_false = false; //variable template.
//To be explained soon

void doStuff(auto x) //notice lack of template keyword!
//This is still function template!
{
  if constexpr(std::same_as<bool, decltype(x)>) {
  }
  else if constexpr(std::is_pointer_v<decltype(x)>) {

  }
  else { static_assert(dependent_false<decltype(x)>, "!"); }
}


int main(int, char**)
{
    doStuff(true); //OK
    int x = 3;
    doStuff(&x); //now it works
}
```


Function templates

SFINAE – Substitution Failure Is Not An Error

SFINAE is also applicable for class templates

```
template<typename T>
constexpr bool dependentFalse = false;

template<typename T>
struct FixMe
{
    void IWontCompile()
    { static_assert(dependentFalse<T>, "");}
};


template struct FixMe<int>;
```

Can you spot the problem here?

https://godbolt.org/z/11YsPb3oY

Function templates

SFINAE – Substitution Failure Is Not An Error

SFINAE is also applicable for class templates

```
template<typename T>
constexpr bool dependentFalse = false;

template<typename T>
struct FixMe
{
    void IWontCompile() requires(false)
    { static_assert(dependentFalse<T>, "");}
};


template struct FixMe<int>;
```

Can you spot the problem here?

This technically is not a fuction template

https://godbolt.org/z/GcMv1jq7G

https://godbolt.org/z/aG71xWbef (pre C++20, not exactly equivalent)

Function templates

SFINAE – Substitution Failure Is Not An Error

SFINAE is also applicable for class templates

```
template<typename T>
constexpr bool dependentFalse = false;

template<typename T>
struct FixMe
{
    template<typename U=T> requires false
    void IWontCompile()
    { static_assert(dependentFalse<U>, "");}
};


template struct FixMe<int>;
```

Can you spot the problem here?

But this is a function template

https://godbolt.org/z/3ddWr7o9j another C++20 way

https://godbolt.org/z/aG71xWbef (pre C++20)

Function templates

Perfect forwarding

Imagine the following example

```
#include <string>

class Person
{
public:
  explicit Person(const std::string& n) : name(n) {}
private:
  std::string name{};
};

int main(int, char**)
{
  Person mike{std::string("mike")};
  std::string j = "john";
  Person john{j};
}
```

What is wrong with it?

Function templates

Perfect forwarding

Imagine the following example

```
#include <string>

class Person
{
public:
  explicit Person(const std::string& n) : name(n) {}
private:
  std::string name{};
};

int main(int, char**)
{
  Person mike{std::string("mike")};
  std::string j = "john";
  Person john{j};
}
```

What is wrong with it?

https://godbolt.org/z/W3nj4EW1n

Function templates

Perfect forwarding

Imagine the following example

```
#include <string>

class Person
{
public:
  explicit Person(const std::string& n) : name(n) {}
private:
  std::string name{};
};

int main(int, char**)
{
  Person mike{std::string("mike")};
  std::string j = "john";
  Person john{j};
}
```

This is wasteful

What is wrong with it?

https://godbolt.org/z/W3nj4EW1n

Function templates

Perfect forwarding

Imagine the following example

```
#include <string>

class Person
{
public:
  explicit Person(const std::string& n) : name(n) {}
private:
  std::string name{};
};

int main(int, char**)
{
  Person mike{std::string("mike")};
  std::string j = "john";
  Person john{j};
}
```

Copy occurs here

This is wasteful

What is wrong with it?

https://godbolt.org/z/W3nj4EW1n

Function templates

Perfect forwarding

Imagine the following example

```
#include <string>

class Person
{
public:
  explicit Person(const std::string& n) : name(n) {}
private:
  std::string name{};
};

int main(int, char**)
{
  Person mike{std::string("mike")};
  std::string j = "john";
  Person john{j};
}
```

Copy occurs here…

…of a temporary string

This is wasteful

What is wrong with it?

https://godbolt.org/z/W3nj4EW1n

Function templates

Perfect forwarding

Imagine the following example

```
#include <string>

class Person
{
public:
  explicit Person(const std::string& n) : name(n) {}
private:
  std::string name{};
};

int main(int, char**)
{
  Person mike{std::string("mike")};
  std::string j = "john";
  Person john{j};
}
```

Copy occurs here...

...of a temporary string

This is wasteful

How to fix this?

https://godbolt.org/z/W3nj4EW1n

Function templates

Perfect forwarding

Imagine the following example

```
#include <string>

class Person
{
public:
  template<typename T>
  explicit Person(T&& n) : name(std::forward<T>(n)) {}
private:
  std::string name{};
};

int main(int, char**)
{
  Person mike{std::string("mike")};
  std::string j = "john";
  Person john{j};
}
```

Copy or move occurs here…

Move of a temporary string

Better now?

Function templates

Perfect forwarding

Imagine the following example

```
#include <string>

class Person
{
public:
  template<typename T>
  explicit Person(T&& n) : name(std::forward<T>(n)) {}
private:
  std::string name{};
};

int main(int, char**)
{
  Person mike{std::string("mike")};
  std::string j = "john";
  Person john{j};
  Person john2{j};

}
```

Copy or move occurs here…

Move of a temporary string

Due to type deducation, template constructor tries to become a copy constructor

Better now?

Not yet. This template tries to instantiate the copy constructor

https://godbolt.org/z/9v1qajq9h

Function templates

Perfect forwarding

Imagine the following example

```
#include <string>

class Person
{
public:
  template<typename T>
  explicit Person(T&& n)
  requires(std::same_as<std::string, std::decay_t<T>>)
  : name(std::forward<T>(n)) {}
private:
  std::string name{};
};

int main(int, char**)
{
  Person mike{std::string("mike")};
  std::string j = "john";
  Person john{j};
  Person john2{j};
}
```

SFINAE again, substitution fails, default copy kicks in

https://godbolt.org/z/nT7EoPqhr

Function templates

Perfect forwarding

Imagine the following example

```
#include <string>

class Person
{
public:
  template<typename T>
  explicit Person(T&& n)
  requires(std::same_as<std::string, std::decay_t<T>>)
  : name(std::forward<T>(n)) {}
private:
  std::string name{};
};

int main(int, char**)
{
  Person mike{std::string("mike")};
  std::string j = "john";
  Person john{j};
  Person john2{j};
}
```

And whas is decay_t?

https://godbolt.org/z/nT7EoPqhr

Function templates

Perfect forwarding

std::decay

Defined in header <type_traits>

```
template< class T >
struct decay;
```
(since C++11)

Performs the type conversions equivalent to the ones performed when passing function arguments by value. Formally:

- If T is "array of U" or reference to it, the member typedef type is U*.
- Otherwise, if T is a function type F or reference to one, the member typedef type is std::add_pointer<F>::type.
- Otherwise, the member typedef type is std::remove_cv<std::remove_reference<T>::type>::type.

If the program adds specializations for std::decay, the behavior is undefined.

Member types

Name	Definition
type	the result of applying the decay type conversions to T

Helper types

```
template< class T >
using decay_t = typename decay<T>::type;
```
(since C++14)

https://en.cppreference.com/w/cpp/types/decay

Function templates

Perfect forwarding

Imagine the following example

```
#include <string>

class Person
{
public:
  template<typename T>
  explicit Person(T&& n)
  requires(std::same_as<std::string, std::decay_t<T>>)
  : name(std::forward<T>(n)) {}
private:
  std::string name{};
};

int main(int, char**)
{
  Person mike{std::string("mike")};
  std::string j = "john";
  Person john{j};
  Person john2{j};
}
```

T is std::string

T is std::string&

https://godbolt.org/z/nT7EoPqhr

Function templates

Perfect forwarding

Imagine the following example

```
#include <string>

class Person
{
public:
  template<typename T>
  explicit Person(T&& n)
  requires(std::same_as<std::string, std::decay_t<T>>)
  : name(std::forward<T>(n)) {}
private:
  std::string name{};
};

int main(int, char**)
{
  Person mike{std::string("mike")};
  std::string j = "john";
  Person john{j};
  Person john2{j};
}
```

T is std::string

T is std::string&

```
(pseudocode)
T&&

std::string  && -> std::string&& //rvalue reference
std::string& && -> std::string&  //lvalue reference
```

https://godbolt.org/z/nT7EoPqhr

Function templates

Perfect forwarding

Imagine the following example

```
#include <string>

class Person
{
public:
  template<typename T>
  explicit Person(T&& n)
  requires(std::same_as<std::string, std::decay_t<T>>)
  : name(std::forward<T>(n)) {}
private:
  std::string name{};
};

int main(int, char**)
{
  Person mike{std::string("mike")};
  std::string j = "john";
  Person john{j};
  Person john2{std::move(j)};
}
```

T is std::string

T is std::string&

T is std::string, value of j is consumed

```
(pseudocode)
T&&

std::string  && -> std::string&& //rvalue reference
std::string& && -> std::string&  //lvalue reference
```

Reference collapsing rules:

& & =&

& && = &

&& & = &

&& && = &&

https://godbolt.org/z/nT7EoPqhr

Function templates

Perfect forwarding

Imagine the following example

```
#include <string>

class Person
{
public:
  template<typename T>
  explicit Person(T&& n)
  requires(std::same_as<std::string, std::decay_t<T>>)
  : name(std::forward<T>(n)) {}
private:
  std::string name{};
};

int main(int, char**)
{
  Person mike{std::string("mike")};
  std::string j = "john";
  Person john{j};
  Person john2{std::move(j)};
}
```

T is std::string

T is std::string&

T is std::string, value of j is consumed

```
(pseudocode)
T&&

std::string  && -> std::string&& //rvalue reference
std::string& && -> std::string&  //lvalue reference
```

Reference collapsing rules:

& & =&

& && = &

&& & = &

&& && = &&

https://godbolt.org/z/nT7EoPqhr

Function templates

Perfect forwarding

Simplified?

```
#include <string>

class Person
{
public:

  explicit Person(std::string n)
  : name(std::move(n)) {}
private:
  std::string name{};
};

int main(int, char**)
{
  Person mike{std::string("mike")};
  std::string j = "john";
  Person john{j};
  Person john2{std::move(j)};
}
```

No template
Pass by value
Value consumed

https://godbolt.org/z/sqr698vc8

Function templates

Perfect forwarding

Simplified?

```
#include <string>

class Person
{
public:

  explicit Person(std::string n)
  : name(std::move(n)) {}
private:
  std::string name{};
};

int main(int, char**)
{
  Person mike{std::string("mike")};
  std::string j = "john";
  Person john{j};
  Person john2{std::move(j)};
}
```

No template

Pass by value

Value consumed

OK, temporary+move

OK, copy on call site

OK, move on call site

https://godbolt.org/z/sqr698vc8

Function templates

Perfect forwarding

+simpler
+no templates
+can be out-of-line (in cpp file)
+type conversions allowed

-exception guarantees (if applicable)
-potentially performance will differ

Simplified?

```
#include <string>

class Person
{
public:

  explicit Person(std::string n)
  : name(std::move(n)) {}
private:
  std::string name{};
};

int main(int, char**)
{
  Person mike{std::string("mike")};
  std::string j = "john";
  Person john{j};
  Person john2{std::move(j)};
}
```

No template
Pass by value
Value consumed

OK, temporary+move
OK, copy on call site
OK, move on call site

https://godbolt.org/z/sqr698vc8

Concepts and constraints

Concept

1. Compile time Type to boolean mapping
2. Can be used to constrain template code

Concepts and constraints

Concepts

```
#include <concepts>

#include <array>
#include <vector>

template<typename T>
concept RandomAccessible = requires (T x, std::size_t adr)
{
    { x[adr] } -> std::same_as<typename T::reference>;
    { std::bool_constant<noexcept(x[adr])>{} } -> std::same_as<std::true_type>;
    { x.at(adr)} -> std::same_as<typename T::reference>;
};


struct SafeDynamicIntArray
{
    std::vector<int> contents{};
    int& operator[](std::size_t pos) { return contents.at(pos);}
    const int& operator[](std::size_t pos) const { return contents.at(pos);}
    int& at(std::size_t pos) { return contents.at(pos);}
    const int& at(std::size_t pos) const { return contents.at(pos);}

    using reference = int&;
};

static_assert(RandomAccessible<std::vector<char>>, "");
static_assert(RandomAccessible<std::array<int, 42>>, "");

static_assert(!RandomAccessible<SafeDynamicIntArray>, "");
```

Expression

Concept that the exception yields

https://godbolt.org/z/Wvf8Y5839

Concepts and constraints

Concepts

Dummy variables

Concept that the expression yields

Expression

```
#include <concepts>

#include <array>
#include <vector>

template<typename T>
concept RandomAccessible = requires (T x, std::size_t adr)
{
    { x[adr] } -> std::same_as<typename T::reference>;
    { std::bool_constant<noexcept(x[adr])>{} } -> std::same_as<std::true_type>;
    { x.at(adr)} -> std::same_as<typename T::reference>;
};


struct SafeDynamicIntArray
{
    std::vector<int> contents{};
    int& operator[](std::size_t pos) { return contents.at(pos);}
    const int& operator[](std::size_t pos) const { return contents.at(pos);}
    int& at(std::size_t pos) { return contents.at(pos);}
    const int& at(std::size_t pos) const { return contents.at(pos);}

    using reference = int&;
};

static_assert(RandomAccessible<std::vector<char>>, "");
static_assert(RandomAccessible<std::array<int, 42>>, "");

static_assert(!RandomAccessible<SafeDynamicIntArray>, "");
```

https://godbolt.org/z/Wvf8Y5839

Concepts and constraints

Concepts

```
#include <concepts>

#include <array>
#include <vector>

template<typename T>
concept RandomAccessible = requires (T x, std::size_t adr)
{
    { x[adr] } -> std::same_as<typename T::reference>;
    { x.at(adr)} -> std::same_as<typename T::reference>;
};


template<typename T>
concept NoexceptRandomAccessible = RandomAccessible<T> and requires(T x, std::size_t adr) {
    { std::bool_constant<noexcept(x[adr])>{} } -> std::same_as<std::true_type>;
};

struct SafeDynamicIntArray
{
    std::vector<int> contents{};
    int& operator[](std::size_t pos) { return contents.at(pos);}
    const int& operator[](std::size_t pos) const { return contents.at(pos);}
    int& at(std::size_t pos) { return contents.at(pos);}
    const int& at(std::size_t pos) const { return contents.at(pos);}

    using reference = int&;
};

static_assert(NoexceptRandomAccessible<std::vector<char>>, "");
static_assert(NoexceptRandomAccessible<std::array<int, 42>>, "");

static_assert(RandomAccessible<SafeDynamicIntArray>, "");
static_assert(!NoexceptRandomAccessible<SafeDynamicIntArray>, "");
```

Note it is a boolean expression

https://godbolt.org/z/hYq5Y6eY7

Concepts and constraints

Constraints

```
//consider the concept from previous slide
static_assert(NoexceptRandomAccessible<std::array<int, 42>>, "");

template<typename T>
requires NoexceptRandomAccessible<T>
void f(const T&) { }

int main(int, char**)
{
  f(std::array<char, 5>{});
  //f(SafeDynamicIntArray{});
}
```

```
template<NoexceptRandomAccessible T>
void f(const T&) { }
```

```
void f(const NoexceptRandomAccessible  auto&) { }
```

https://godbolt.org/z/5Gda8vhEG

Advanced topics, caveats, pitfalls

CRTP and compile-time polymorphism

```
template<typename T>
class Logger
{
public:
    void print(const std::string& x) const {
static_cast<T*>(this)->print(x); }
private:
    Logger() = default;
    friend T;
};
```


Advanced topics, caveats, pitfalls

CRTP and compile-time polymorphism

```
template<typename T>
class Logger
{
public:
    void print(const std::string& x) const {
static_cast<T*>(this)->print(x); }
private:
    Logger() = default;
    friend T;
};

struct DebugLogger : Logger<DebugLogger>
{
    DebugLogger() = default;
    void print(const std::string& x) const
    { std::cout << "DEBUG " << x << '\n'; }
};
```


Advanced topics, caveats, pitfalls

CRTP and compile-time polymorphism

```
template<typename T>
class Logger
{
public:
    void print(const std::string& x) const {
static_cast<T*>(this)->print(x); }
private:
    Logger() = default;
    friend T;
};

struct DebugLogger : Logger<DebugLogger>
{
    DebugLogger() = default;
    void print(const std::string& x) const
    { std::cout << "DEBUG " << x << '\n'; }
};

struct WarningLogger : Logger<WarningLogger>
{
    WarningLogger() = default;
    void print(const std::string& x) const
    { std::cout << "WARNING " << x << '\n'; }
};
```

Constructor and friend not strictly needed, but enforce the right template argument

https://godbolt.org/z/M9rWcj97P

https://godbolt.org/z/a37xdr396

Advanced topics, caveats, pitfalls

Type traits might be lying

```
struct Dog
{
    void bark();
};
struct Cat
{
    void meow();
};
template<typename T>
struct AnimalWrapper
{
    T* contents;
    explicit AnimalWrapper(T& t) : contents(&t){}
    void makeSound() { contents->meow(); }

};
template<typename T>
concept LoudAnimal = requires(T animal) {
    { animal.makeSound() };
};
```

https://godbolt.org/z/716s1zzqe

Why does this build? →

```
int main(int, char**)
{
    Dog d;
    AnimalWrapper<Dog> aw{d};
    static_assert(LoudAnimal<AnimalWrapper<Dog>>);
    // aw.makeSound();
}
```


Advanced topics, caveats, pitfalls

Type traits might be lying

```
struct Dog
{
    void bark();
};

struct Cat
{
    void meow();
};


template<typename T>
struct AnimalWrapper
{
    T* contents;
    explicit AnimalWrapper(T& t) : contents(&t){}

    void makeSound() { contents->meow(); }


};

int main(int, char**)
{
    Dog d;
    AnimalWrapper<Dog> aw{d};
    // aw.makeSound();
}
```

Function body fails to compile

But the signature is visible

Advanced topics, caveats, pitfalls

Type traits might be lying

```
struct Dog
{
    void bark();
};

struct Cat
{
    void meow();
};


template<typename T>
struct AnimalWrapper
{
    T* contents;
    explicit AnimalWrapper(T& t) : contents(&t){}

    void makeSound() requires (requires {{contents->meow() };} ){ contents.meow(); }
    void makeSound() requires (requires {{contents->bark() };} ){ contents->bark(); }

};

int main(int, char**)
{
    Dog d;
    AnimalWrapper<Dog> aw{d};
    // aw.makeSound();
}
```

https://godbolt.org/z/r8W8fKqzq

Inline concept definition

SFINAE

Advanced topics, caveats, pitfalls

Type traits might be lying

```
struct Dog
{
    void bark();
};

struct Cat
{
    void meow();
};

template<typename T>
constexpr bool dependentFalse = false;

template<typename T>
struct AnimalWrapper
{
    T* contents;
    explicit AnimalWrapper(T& t) : contents(&t){}

    void makeSound()
    {
        if constexpr (requires { {contents->bark()}; }) {
            contents->bark();
        } else if constexpr (requires { {contents->meow()}; }) {
            contents->meow();
        } else {
            static_assert(dependentFalse<T>, "should not reach here");
        }
    }
};
```

https://godbolt.org/z/r8W8fKqzq

Advanced topics, caveats, pitfalls

Fun with parameter packs

```
#include <tuple>
#include <iostream>

template<typename ...T1, typename ...T2>
void printTupleSizes(const std::tuple<T1...>&, const std::tuple<T2...>&)
{
    std::cout << sizeof...(T1) << ' ' << sizeof...(T2) << '\n';
}

int main(int, char**)
{
    std::tuple<int, char, double> t1{};
    std::tuple<float, int, int, int, int> t2{};
    printTupleSizes(t1, t2);
}
```

https://godbolt.org/z/EhhMdqMjW

Advanced topics, caveats, pitfalls

Fun with parameter packs

```
#include <tuple>
#include <iostream>

template<typename ...T1, typename ...T2>
void printTupleSizes(const std::tuple<T1...>&, const std::tuple<T2...>&)
{
    std::cout << sizeof...(T1) << ' ' << sizeof...(T2) << '\n';
}

int main(int, char**)
{
    std::tuple<int, char, double> t1{};
    std::tuple<float, int, int, int, int> t2{};
    printTupleSizes(t1, t2);
}
```

https://godbolt.org/z/EhhMdqMjW

Advanced topics, caveats, pitfalls

Fun with parameter packs

```cpp
#include <tuple>
#include <iostream>
#include <utility>

template<typename TUPLE>
void printEvenIndices(const TUPLE& t)
{
    []<std::size_t... I>(const TUPLE& t, std::index_sequence<I...>)
    {

        ((I % 2 == 0
            ? std::cout <<  std::get<I>(t) << ' '
            : std::cout << ' '), ...);
    }(t, std::make_index_sequence<std::tuple_size_v<TUPLE>>{});
}
```

```cpp
int main(int, char**)
{
    std::tuple<int, int, int, int, int> t1{1, 2, 3, 4, 5};
    printEvenIndices(t1);

}
```

https://godbolt.org/z/9KP9M74f3

Advanced topics, caveats, pitfalls

Fun with parameter packs

```cpp
template<auto> struct WrapVal{};
template<auto... Values> struct WrapSeq{};

template<auto...SA, auto...SB>
constexpr WrapSeq<SA..., SB...> operator+(WrapSeq<SA...>, WrapSeq<SB...>)
{
    return{};
}
template<auto Val> constexpr auto filterSingleEven(WrapVal<Val>)
{
    if constexpr(Val % 2 == 0) {
        return WrapSeq<Val>{};
    } else {
        return WrapSeq<>{};
    }
}

template<auto ...Vals> constexpr auto filterSeqEven(WrapSeq<Vals...>)
{
    return (filterSingleEven(WrapVal<Vals>{}) + ...);
}
```

```cpp
template<std::size_t...I>
constexpr auto toCustomSeq(std::index_sequence<I...>)
{
    return WrapSeq<I...>{};
}

template<typename TUPLE>
auto printEvenIndices(const TUPLE& t)
{
    []<auto...IDX>(const TUPLE& t, WrapSeq<IDX...>) {
        ((std::cout << std::get<IDX>(t) << ' '), ...);
    }(t, filterSeqEven(toCustomSeq(
std::make_index_sequence<std::tuple_size_v<TUPLE>>{})));
    std::cout << '\n';
}

int main(int, char**)
{
    std::tuple<int, int, int, int, int> t1{1, 2, 3, 4, 5};
    printEvenIndices(t1);
}
```

https://godbolt.org/z/4nYn65f58

Advanced topics, caveats, pitfalls

How does auto work?

```
template<typename T>
void f1(T&& x) { }
```

```
void f1(auto&& x) { }
```

```
template<typename...Args>
void f1(Args...&& x) { }
```

```
void f1(auto...&& args) { }
```

Advanced topics, caveats, pitfalls

How does auto work?

```
template<typename T>
void f1(T&& x) { }
```

```
void f1(auto&& x) { }
```

```
template<typename...Args>
void f1(Args...&& x) { }
```

```
void f1(auto...&& args) { }
```

OK, what about perferct forwarding?

Advanced topics, caveats, pitfalls

How does auto work?

```
template<typename T>
void f1(T&& x) { }
```

```
void f1(auto&& x) { }
```

```
template<typename...Args>
void f1(Args...&& x) { }
```

```
void f1(auto...&& args) { }
```

OK, what about perferct forwarding?

```
void f1(auto&& x)
{
  inner(std::forward<decltype(x)>(x));
}
```

```
void f1(auto&& x)
{
  inner(std::forward<decltype(args)>(args)...);
}
```


Advanced topics, caveats, pitfalls

How does auto work?

```
template<typename T>
void f1(T&& x) { }
```

```
void f1(auto&& x) { }
```

```
template<typename...Args>
void f1(Args...&& x) { }
```

```
void f1(auto...&& args) { }
```

OK, what about perferct forwarding?

```
void f1(auto&& x)
{
  inner(std::forward<decltype(x)>(x));
}
```

```
void f1(auto&& x)
{
  inner(std::forward<decltype(args)>(args)...);
}
```

What? And what is decltype?

Advanced topics, caveats, pitfalls

How does auto work?

```
template<typename T>
void f1(T&& x) { }
```

```
void f1(auto&& x) { }
```

```
template<typename...Args>
void f1(Args...&& x) { }
```

```
void f1(auto...&& args) { }
```

OK, what about perferct forwarding?

```
void f1(auto&& x)
{
  inner(std::forward<decltype(x)>(x));
}
```

```
void f1(auto&& x)
{
  inner(std::forward<decltype(args)>(args)...);
}
```

What? And what is decltype?

```
void f1(auto&& x)
{
  inner(std::forward<decltype(x)>(x));
}
```

Replace auto with T. It can be deduced as T& or T (+ const/volatile)

decltype gets the type of expression/variable

This will need a small explanation

Advanced topics, caveats, pitfalls

Perfect forwarding explained

```
decltype(std::forward<int>) -> int&&
decltype(std::foward<int&> -> int&
decltype(std::foward<int&&> -> int&&
```

What if std::forward just added && to each type?

Advanced topics, caveats, pitfalls

Perfect forwarding explained

```
decltype(std::forward<int>) -> int&&
decltype(std::foward<int&> -> int&
decltype(std::foward<int&&> -> int&&
```

What if std::forward just added && to each type?

```
decltype(std::forward<int>) -> int && -> int&&
decltype(std::foward<int&> -> int& && -> int&
decltype(std::foward<int&&> -> int&& && -> int&&
```

Advanced topics, caveats, pitfalls

Perfect forwarding explained

```
decltype(std::forward<int>) -> int&&
decltype(std::foward<int&> -> int&
decltype(std::foward<int&&> -> int&&
```

What if std::forward just added && to each type?

```
decltype(std::forward<int>) -> int && -> int&&
decltype(std::foward<int&> -> int& && -> int&
decltype(std::foward<int&&> -> int&& && -> int&&
```

Actually, it does!

Advanced topics, caveats, pitfalls

Dependent types and templates

```
template<typename T>
struct Something
{
    using SuppliedType = T;

    template<typename U>
    void func(U) { }

    template<typename U>
    void func2() { }


};

template<typename T>
void doSomething(T)
{

    Something<T> s;
    typename Something<T>::SuppliedType x = 42;
    s.func('a');
    s.template func2<char>();
}


int main(int, char**)
{
    doSomething(3);
}
```

Dependent (on T parameter) type name

Function call from function template, deduced arguments

Dependent (on T parameter) type name function call, with explicitly given template parameters

https://godbolt.org/z/fhE4qznee

Further learning material

Recommended

1. Modern C++ Design: Generic Programming and Design Patterns, Alexandrescu A., Lafferty D.
2. C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond, Abrahams D., Gurtovoy A.
3. Move semantics: The Complete Guide, Josuttis N.
4. The Nightmare of Initialization in C++, Josuttis N. https://www.youtube.com/watch?v=7DTlWPgX6zs

Questions?

NOKIA