Not So Tiny Task №11 (0.5 point)

Calculate Nth prime number in compile-time! (without constexpr)

System Programming with C++

Templates implementation, SFINAE, std::conditional, std::enable_if

Further directions

- More detailed templates implementation + meta-programming + compile-time evaluation

- Is it really necessary to specify a type explicitly each time for instantiation?

- Variadic templates and requires

Two-phase name lookup

```
template<typename Derived>
class Person {
public:
    void print() {
        static_cast<Derived*>(this)->print_impl();
    }
};
```

Are there any mistakes in these code?

Will it be compiled?

Two-phase name lookup

```
template<typename Derived>
class Person {
public:
    void print() {
        static_cast<Derived*>(this)->print_impl();
    }
};
```

Are there any mistakes in these code?

Not yet.

Will it be compiled?

Yes.

Two-phase name lookup

```
template<typename Derived>
class Person {
public:
    void print() {
        static_cast<Derived*>(this)->print_impl();
    }
};
```

Are there any mistakes in these code?
Not yet.

Will it be compiled?
Yes.

Well, actually it depends on what will be placed instead of "Derived".

Two-phase name lookup

```
template<typename Derived>
class Person {
public:
    void print() {
        foo231();
        static_cast<Derived*>(this)->print_impl();
    }
};
```

Are there any mistakes in these code?

Will it be compiled?

Two-phase name lookup

```
template<typename Derived>
class Person {
public:
    void print() {
        foo231();
        static_cast<Derived*>(this)->print_impl();
    }
};
```

Are there any mistakes in these code?
Yes.

Will it be compiled?
No.

Two-phase name lookup

```
template<typename Derived>
class Person {
public:
    void print() {
        foo231();
        static_cast<Derived*>(this)->print_impl();
    }
};
```

Are there any mistakes in these code?
Yes.

Will it be compiled?
No.

error: there are no arguments to 'foo231' that depend on a template parameter, so a declaration of 'foo231' must be available

Two-phase name lookup

```
template<typename Derived>
class Person {
public:
    void print() {
        foo231();
        static_cast<Derived*>(this)->print_impl();
    }
};
```

```
error: there are no arguments to 'foo231' that
depend on a template parameter, so a declaration
of 'foo231' must be available
```

Are there any mistakes in these code?
Yes.

Will it be compiled?
No.

This time it doesn’t matter what will be used instead of Derived, we just do not have foo231!

Two-phase name lookup

```
template<typename Derived>
class Person {
public:
    void print() {
        foo231();
        static_cast<Derived*>(this)->print_impl();
    }
};
```

error: there are no arguments to 'foo231' that depend on a template parameter, so a declaration of 'foo231' must be available

There are 2 phases of name lookup:

1) Before instantiation of templates => check only independent names

2) After instantiation of templates => check the rest (template names)

Two-phase name lookup

How to make foo231
dependent on Derived?

```
template<typename Derived>
class Person {
public:
    void print() {
        foo231();
        static_cast<Derived*>(this)->print_impl();
    }
};
```

error: there are no arguments to 'foo231' that depend on a template parameter, so a declaration of 'foo231' must be available

There are 2 phases of name lookup:

1) Before instantiation of templates => check only independent names

2) After instantiation of templates => check the rest (template names)

Two-phase name lookup

```
template<typename Derived>
class Person {
public:
    void print() {
        Derived::foo231();
        static_cast<Derived*>(this)->print_impl();
    }
};
```

Now it compiles without problems.

There are 2 phases of name lookup:

1) Before instantiation of templates => check only independent names

2) After instantiation of templates => check the rest (template names)

Two-phase name lookup

```
template<typename Derived>
class Person {
public:
    void print() {
        Derived::foo231();
        static_cast<Derived*>(this)->print_impl();
    }
};
```

Now it compiles without problems. Sounds good!
But actually...

There are 2 phases of name lookup:

1) Before instantiation of templates => check only independent names

2) After instantiation of templates => check the rest (template names)

Two-phase name lookup

```
template<typename Derived>
class Person {
public:
    void print() {
        Derived::foo231();
        static_cast<Derived*>(this)->print_impl();
    }
};
```

Now it compiles without problems. Sounds good!
But actually...

There are 2 phases of name lookup:

1) Before instantiation of templates => check only independent names

2) After instantiation of templates => check the rest (template names)

Two-phase name lookup

```
template<typename T>
class Base {
public:
   void exit() {}
};
```

Two-phase name lookup

```
template<typename T>
class Base {
public:
   void exit() {}
};


template<typename T>
class Derived: Base<T> {
public:
   void foo() {
       exit();
   }
};
```

Two-phase name lookup

```
template<typename T>
class Base {
public:
   void exit() {}
};


template<typename T>
class Derived: Base<T> {
public:
   void foo() {
       exit();
   }
};
```

Will it compile?

Two-phase name lookup

```cpp
template<typename T>
class Base {
public:
   void exit() {}
};


template<typename T>
class Derived: Base<T> {
public:
   void foo() {
       exit();
   }
};
```

Will it compile?

```
main.cpp: In member function 'void Derived<T>::foo()':
main.cpp:979:13: error: too few arguments to function
                                               'void exit(int)'
  979 |          exit();
      |          ~~~~^~
```

Two-phase name lookup

```
template<typename T>
class Base {
public:
   void exit() {}
};


template<typename T>
class Derived: Base<T> {
public:
   void foo() {
       exit();
   }
};
```

Will it compile?

```
main.cpp: In member function 'void Derived<T>::foo()':
main.cpp:979:13: error: too few arguments to function
                                       'void exit(int)'
  979 |         exit();
      |         ~~~~^~

…mingw/x86_64-w64-mingw32/include/process.h:42:32:
note: declared here
   42 |   void __cdecl __MINGW_NOTHROW exit(int _Code)
__MINGW_ATTRIB_NORETURN;
      |                               ^~~~
```

Two-phase name lookup

```
template<typename T>
class Base {
public:
   void exit() {}
};


template<typename T>
class Derived: Base<T> {
public:
   void foo() {
       exit();
   }
};
```

Will it compile?

No, because exit() looks like an independent name for the compiler.

Two-phase name lookup

```
template<typename T>
class Base {
public:
    void exit() {}
};


template<typename T>
class Derived: Base<T> {
public:
    void foo() {
        exit();
    }
};
```

Will it compile?

No, because exit() looks like an independent name for the compiler.

So, it tries to lookup it during 1st phase. And the best candidate is exit from standard lib.

Two-phase name lookup

```cpp
template<typename T>
class Base {
public:
    void exit() {}
};
```

```cpp
template<typename T>
class Derived: Base<T> {
public:
    void foo() {
        exit();
    }
};
```

How to make it dependent?

Will it compile?

No, because exit() looks like an independent name for the compiler.

So, it tries to lookup it during 1st phase. And the best candidate is exit from standard lib.

Two-phase name lookup

```
template<typename T>
class Base {
public:
    void exit() {}
};


template<typename T>
class Derived: Base<T> {
public:
    void foo() {
        Base<T>::exit();
    }
};
```

How to make it dependent? ✓

Will it compile?

No, because exit() looks like an independent name for the compiler.

So, it tries to lookup it during 1st phase. And the best candidate is exit from standard lib.

Two-phase name lookup

```
template<typename T>
class Base {
public:
   void exit() {}
};
```

```
template<typename T>
class Derived: Base<T> {
public:
   void foo() {
       this->exit();
   }
};
```

How to make it dependent?

Will it compile?

No, because exit() looks like an independent name for the compiler.

So, it tries to lookup it during 1st phase. And the best candidate is exit from standard lib.

Two-phase name lookup

```
template<typename T>
class Base {
public:
   void exit() {}
};

template<typename T>
class Derived: Base<T> {
public:
   void foo() {
      this->exit();
   }
};
```

How to make it dependent?

Nice usage of explicit this->

Will it compile?

No, because exit() looks like an independent name for the compiler.

So, it tries to lookup it during 1st phase. And the best candidate is exit from standard lib.

Two-phase name lookup

```
template<typename T>
class Baz {
public:
    class Bar {};
};
```

Two-phase name lookup

```
template<typename T>
class Baz {
public:
    class Bar {};
};

template<typename T>
void foo() {
    T::Bar* pointer;
}

foo<Baz>();
```

Two-phase name lookup

```
template<typename T>
class Baz {
public:
    class Bar {};
};

template<typename T>
void foo() {
    T::Bar* pointer;
}

foo<Baz>();
```

Will it compile?

Two-phase name lookup

```
template<typename T>
class Baz {
public:
    class Bar {};
};

template<typename T>
void foo() {
    T::Bar* pointer;
}

foo<Baz>();
```

Will it compile?
No.

```
main.cpp:993:13: error: 'pointer' was
not declared in this scope;
  993 |     T::Bar* pointer;
```

Two-phase name lookup

```
template<typename T>
class Baz {
public:
    class Bar {};
};

template<typename T>
void foo() {
    T::Bar* pointer;
}

foo<Baz>();
```

Will it compile?
No.

```
main.cpp:993:13: error: 'pointer' was
not declared in this scope;
  993 |     T::Bar* pointer;
```

But why? I'm just declaring a pointer

Two-phase name lookup

```
template<typename T>
class Baz {
public:
    class Bar {};
};

template<typename T>
void foo() {
    T::Bar* pointer;
}

foo<Baz>();
```

Will it compile?

No.

```
main.cpp:993:13: error: 'pointer' was
not declared in this scope;
  993 |     T::Bar* pointer;
```

What do we know about this T during 1st phase of name lookup?

Two-phase name lookup

```
template<typename T>
class Baz {
public:
    class Bar {};
};

template<typename T>
void foo() {
    T::Bar* pointer;
}

foo<Baz>();
```

Will it compile?
No.

```
main.cpp:993:13: error: 'pointer' was
not declared in this scope;
  993 |     T::Bar* pointer;
```

What do we know about this T during 1st phase of name lookup? Well, it is a type and it has Bar inside.

Two-phase name lookup

```
template<typename T>
class Baz {
public:
    class Bar {};
};

template<typename T>
void foo() {
    T::Bar* pointer;
}

foo<Baz>();
```

Will it compile?
No.

```
main.cpp:993:13: error: 'pointer' was
not declared in this scope;
  993 |      T::Bar* pointer;
```

What do we know about this T during 1st phase of name lookup? Well, it is a type and it has Bar inside. But what is Bar? Nested class? Or maybe static field?

Two-phase name lookup

```
template<typename T>
class Baz {
public:
   class Bar {};
};

template<typename T>
void foo() {
   T::Bar* pointer;
}

foo<Baz>();
```

Will it compile?
No.

```
main.cpp:993:13: error: 'pointer' was
not declared in this scope;
  993 |     T::Bar* pointer;
```

What do we know about this T during 1st phase of name lookup? Well, it is a type and it has Bar inside. But what is Bar? Nested class? Or maybe static field?
It will be clear only during 2nd phase!

Two-phase name lookup

```
template<typename T>
class Baz {
public:
   class Bar {};
};

template<typename T>
void foo() {
   T::Bar* pointer;
}

foo<Baz>();
```

Will it compile?
No.

```
main.cpp:993:13: error: 'pointer' was
not declared in this scope;
  993 |     T::Bar* pointer;
```

What do we know about this T during 1st phase of name lookup? Well, it is a type and it has Bar inside. But what is Bar? Nested class? Or maybe static field? It will be clear only during 2nd phase!

Two-phase name lookup

```
template<typename T>
class Baz {
public:
    class Bar {};
};

template<typename T>
void foo() {
    T::Bar* pointer;
}

foo<Baz>();
```

Will it compile?

No.

```
main.cpp:993:13: error: 'pointer' was
not declared in this scope;
  993 |      T::Bar* pointer;
```

If compiler things that T::Bar is static field, than what is *?

Two-phase name lookup

```
template<typename T>
class Baz {
public:
    class Bar {};
};

template<typename T>
void foo() {
    T::Bar* pointer;
}

foo<Baz>();
```

Will it compile?
No.

```
main.cpp:993:13: error: 'pointer' was
not declared in this scope;
  993 |     T::Bar* pointer;
```

If compiler things that T::Bar is static field, than what is *? Multiplication! With something undefined (pointer). So, compilation error.

Two-phase name lookup

```
template<typename T>
class Baz {
public:
    class Bar {};
};

template<typename T>
void foo() {
    typename T::Bar* pointer;
}

foo<Baz>();
```

Will it compile?
No.

```
main.cpp:993:13: error: 'pointer' was
not declared in this scope;
  993 |     T::Bar* pointer;
```

If compiler things that T::Bar is static field, than what is *? Multiplication! With something undefined (pointer). So, compilation error. How to fix?

Two-phase name lookup

```
template<typename T>
class Baz {
public:
    class Bar {};
};

template<typename T>
void foo() {
    typename T::Bar* pointer;
}

foo<Baz>();
```

Will it compile?
Yes! ✓

Two-phase name lookup

```
template<typename T>
class Baz {
public:
    template<typename U>
    void foo() {}
};
```

Two-phase name lookup

```
template<typename T>
class Baz {
public:
    template<typename U>
    void foo() {}
};

template<typename T>
void foo321() {
    Baz<T> baz;
    baz.foo<T>();
}
```

Will it compile?

Two-phase name lookup

```
template<typename T>
class Baz {
public:
    template<typename U>
    void foo() {}
};

template<typename T>
void foo321() {
    Baz<T> baz;
    baz.foo<T>();
}
```

Will it compile?
No.

```
main.cpp: In function 'void foo321()':
main.cpp:993:14: error: expected
primary-expression before '>' token
  993 |     baz.foo<T>();
      |               ^
main.cpp:993:16: error: expected
primary-expression before ')' token
  993 |     baz.foo<T>();
      |                 ^
```

Two-phase name lookup

```
template<typename T>
class Baz {
public:
   template<typename U>
   void foo() {}
};

template<typename T>
void foo321() {
   Baz<T> baz;
   baz.template foo<T>();
}
```

Will it compile? ✓
Yes.

Ambiguous situation is resolved with `template` keyword.

Two-phase name lookup

```cpp
template<typename T>
class Baz {
public:
    template<typename U>
    void foo() {}
};
```

Will it compile? ✓
Yes.

Ambiguous situation is resolved with `template` keyword.

Funny fact: modern IDEs could catch this problem before compilation.

Takeaways

- 2 phase names lookup: be ready for ambiguous situations and don't be afraid of `template` and `typename` in unusual context

Substitution failure

Substitution failure

```
int negate(int i) {
  return -i;
}
```

Substitution failure

```
int negate(int i) {
   return -i;
}

template <typename T>
T negate(const T& t) {
   typename T::value_type result = -t();
   return result;
}
```

Substitution failure

```
int negate(int i) {
   return -i;
}

template <typename T>
T negate(const T& t) {
   typename T::value_type result = -t();
   return result;
}


int main() {
   return negate(42.0);
}
```

Will it compile?

Substitution failure

```
int negate(int i) {
   return -i;
}

template <typename T>
T negate(const T& t) {
   typename T::value_type result = -t();
   return result;
}


int main() {
   return negate(42.0);
}
```

```
Will it compile?
No.
```

Substitution failure

```
int negate(int i) {
   return -i;
}

template <typename T>
T negate(const T& t) {
   typename T::value_type result = -t();
   return result;
}


int main() {
   return negate(42.0);
}
```

Will it compile?
No.

This function won, it was chosen by the compiler among two negate functions

Substitution failure

```
int negate(int i) {
   return -i;
}

template <typename T>
T negate(const T& t) {
   typename T::value_type result = -t();
   return result;
}


int main() {
   return negate(42.0);
}
```

```
Will it compile?
No.
```

This function won, it was chosen by the compiler among two negate functions

Nothing bad happened during 1st phase.

Substitution failure

```
int negate(int i) {
   return -i;
}

template <typename T>
T negate(const T& t) {
   typename T::value_type result = -t();
   return result;
}


int main() {
   return negate(42.0);
}
```

```
Will it compile?
No.
```

This function won, it was chosen by the compiler among two negate functions

Nothing bad happened during 1st phase.

But during 2nd phase (after substitution) strange constructions appeared:

double::value_type
-double()

How to fix (without changing the usage)?
We actually wanted negate(int) to win!

Substitution failure

```
int negate(int i) {
   return -i;
}

template <typename T>
T negate(const T& t) {
   typename T::value_type result = -t();
   return result;
}


int main() {
   return negate(42.0);
}
```

Will it compile?
No.

This function won, it was chosen by the compiler among two negate functions

Nothing bad happened during 1st phase.

But during 2nd phase (after substitution) strange constructions appeared:

double::value_type
-double()

Substitution failure

How to fix (without changing the usage)?
We actually wanted negate(int) to win!

We can specialize negate for doubles,
but what else?

```
int negate(int i) {
    return -i;
}

template <typename T>
T negate(const T& t) {
    typename T::value_type result = -t();
    return result;
}


int main() {
    return negate(42.0);
}
```

```
Will it compile?
No.
```

This function won, it was chosen by the compiler among two negate functions

Nothing bad happened during 1st phase.

But during 2nd phase (after substitution) strange constructions appeared:

double::value_type
-double()

Substitution failure

```
int negate(int i) {
   return -i;
}

template <typename T>
typename T::value_type negate(const T& t) {
   typename T::value_type result = -t();
   return result;
}


int main() {
   return negate(42.0);
}
```

```
Will it compile?
```

Substitution failure

```
int negate(int i) {
   return -i;
}

template <typename T>
typename T::value_type negate(const T& t) {
   typename T::value_type result = -t();
   return result;
}


int main() {
   return negate(42.0);
}
```

Will it compile? ✓
Yes!

Substitution failure

```
int negate(int i) {
   return -i;
}

template <typename T>
typename T::value_type negate(const T& t) {
   typename T::value_type result = -t();
   return result;
}


int main() {
   return negate(42.0);
}
```

Will it compile?
Yes! ✓

This this function loose! Compiler choose another negate.

Substitution failure

```
int negate(int i) {
   return -i;
}

template <typename T>
typename T::value_type negate(const T& t) {
   typename T::value_type result = -t();
   return result;
}


int main() {
   return negate(42.0);
}
```

Will it compile?
Yes!

✓

This this function loose! Compiler choose another negate.

Because even declaration of this function is enough to understand: it doesn't suit this call.

Substitution failure

```
int negate(int i) {
   return -i;
}

template <typename T>
typename T::value_type negate(const T& t) {
   typename T::value_type result = -t();
   return result;
}


int main() {
   return negate(42.0);
}
```

```
Will it compile?
Yes!
```

✓

This this function loose! Compiler choose another negate.

Because even declaration of this function is enough to understand: it doesn't suit this call. So, substitution of double into this negate function failed.

Substitution failure

```
int negate(int i) {
    return -i;
}

template <typename T>
typename T::value_type negate(const T& t) {
    typename T::value_type result = -t();
    return result;
}


int main() {
    return negate(42.0);
}
```

```
Will it compile?
Yes!
```

✓

This this function loose! Compiler choose another negate.

Because even declaration of this function is enough to understand: it doesn't suit this call. So, substitution of double into this negate function failed.

But it's good news! Because now we can find better function.

Substitution failure is not an error (SFINAE)

```
int negate(int i) {
   return -i;
}

template <typename T>
typename T::value_type negate(const T& t) {
   typename T::value_type result = -t();
   return result;
}


int main() {
   return negate(42.0);
}
```

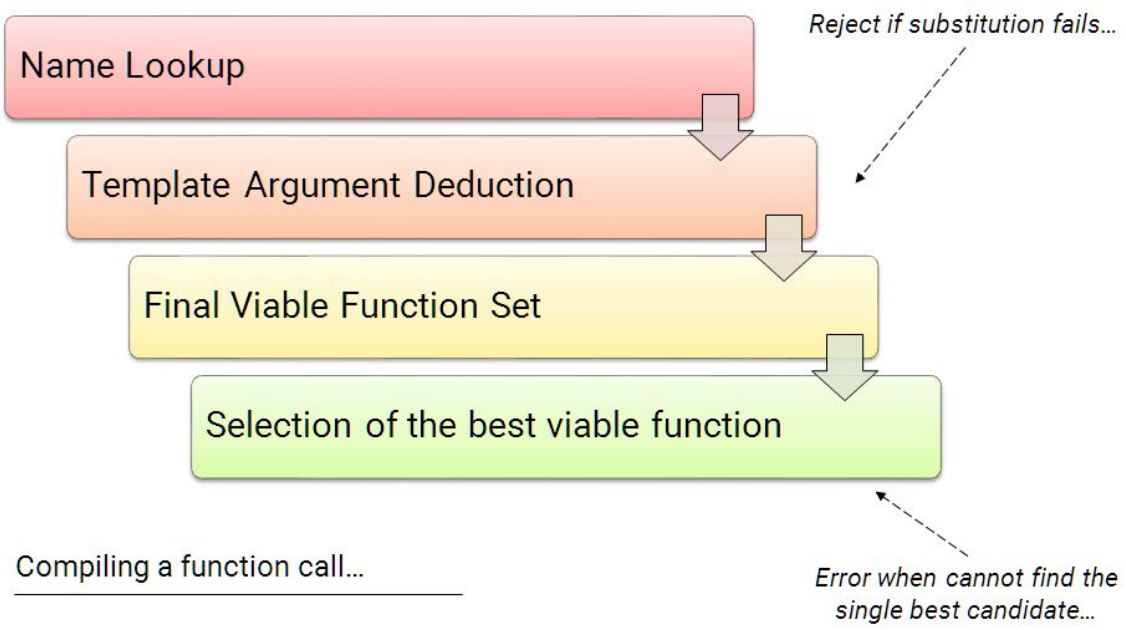
```
Will it compile?
Yes!
```
✓

This this function loose! Compiler choose another negate.

Because even declaration of this function is enough to understand: it doesn't suit this call. So, substitution of double into this negate function failed.

But it's good news! Because now we can find better function.

Substitution failure is not an error (SFINAE)



https://www.cppstories.com/2016/02/notes-on-c-sfinae/

Substitution failure is not an error (SFINAE)

Very simple idea: if some invalid type appears during substitution of actually type to template function, it is not a problem.

Substitution failure is not an error (SFINAE)

Very simple idea: if some invalid type appears during substitution of actually type to template function, it is not a problem.

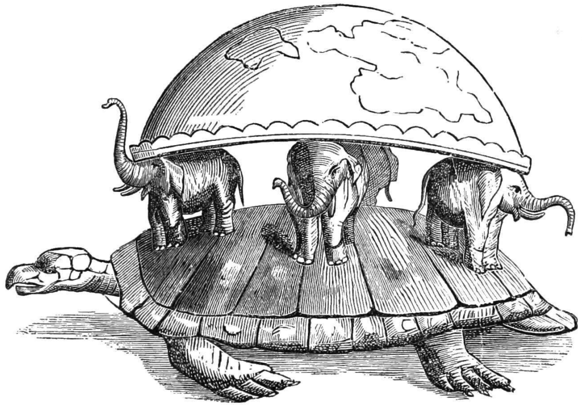
In such case, compiler should just keep searching more suitable overloaded function.

Substitution failure is not an error (SFINAE)

Very simple idea: if some invalid type appears during substitution of actually type to template function, it is not a problem.

In such case, compiler should just keep searching more suitable overloaded function.

So many things in C++ are based on this thing.



Takeaways

- 2 phase names lookup: be ready for ambiguous situations and don’t be afraid of `template` and `typename` in unusual context

- SFINAE as a basic part of names lookup

integral_constant

Task: let's define a separate (wrapper) class for some instance of a class.

integral_constant

Task: let's define a separate (wrapper) class for some instance of a class.

```
template <typename T, T v>
struct integral_constant {
    static const T value = v;




};
```

integral_constant

Task: let’s define a separate (wrapper) class for some instance of a class.

```
template <typename T, T v>
struct integral_constant {
   static const T value = v;
   typedef T value_type;




};
```

integral_constant

Task: let's define a separate (wrapper) class for some instance of a class.

```
template <typename T, T v>
struct integral_constant {
   static const T value = v;
   typedef T value_type;

   operator value_type() const {
       return v;
   }
};
```

integral_constant

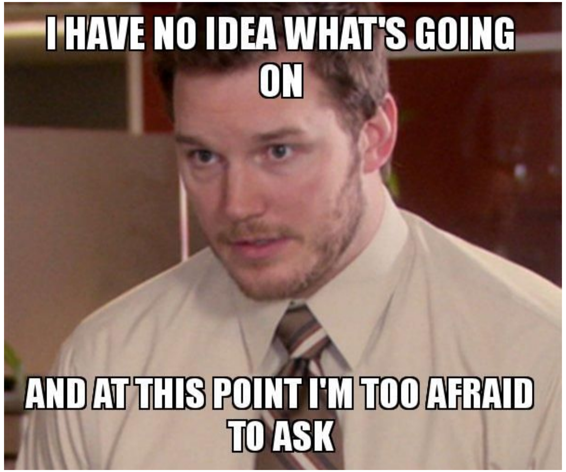
Task: let's define a separate (wrapper) class for some instance of a class.

```
template <typename T, T v>
struct integral_constant {
    static const T value = v;
    typedef T value_type;

    operator value_type() const {
        return v;
    }
};

using ic42 = integral_constant<int, 42>;
int n = 13 * ic42{};
```

integral_constant

Task: let's define a separate (wrapper) class for some instance of a class.

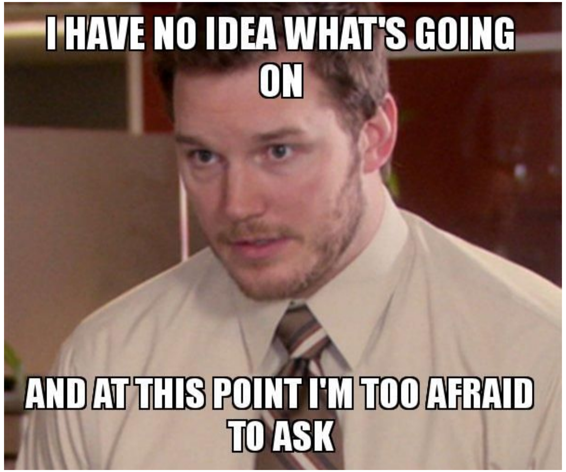
```
template <typename T, T v>
struct integral_constant {
    static const T value = v;
    typedef T value_type;

    operator value_type() const {
        return v;
    }
};

using ic42 = integral_constant<int, 42>;
int n = 13 * ic42{};
```



integral_constant

Task: let's define a separate (wrapper) class for some instance of a class.

```
template <typename T, T v>
struct integral_constant {
    static const T value = v;
    typedef T value_type;
    typedef integral_constant type;
    operator value_type() const {
        return v;
    }
};

using ic42 = integral_constant<int, 42>;
int n = 13 * ic42{};
```



integral_constant

Task: make a checker that takes two template type arguments and check whether they are the same or not.

integral_constant

Task: make a checker that takes two template type arguments and check whether they are the same or not.

```
is_same: T, U -> bool
```

integral_constant

Task: make a checker that takes two template type arguments and check whether they are the same or not.

```
using true_type = integral_constant<bool, true>;
using false_type = integral_constant<bool, false>;
```

integral_constant

Task: make a checker that takes two template type arguments and check whether they are the same or not.

```
using true_type = integral_constant<bool, true>;
using false_type = integral_constant<bool, false>;

template <typename T, typename U>
struct is_same: false_type {};
```

integral_constant

Task: make a checker that takes two template type arguments and check whether they are the same or not.

```
using true_type = integral_constant<bool, true>;
using false_type = integral_constant<bool, false>;

template <typename T, typename U>
struct is_same: false_type {};

template <typename T>
struct is_same<T, T>: true_type {};
```

integral_constant

Task: make a checker that takes two template type arguments and check whether they are the same or not.

```
using true_type = integral_constant<bool, true>;
using false_type = integral_constant<bool, false>;

template <typename T, typename U>
struct is_same: false_type {};

template <typename T>
struct is_same<T, T>: true_type {};

std::cout << is_same<int, int>::value << std::endl;
std::cout << is_same<int, Matrix>::value << std::endl;
std::cout << is_same<Matrix, Matrix>::value << std::endl;
```

integral_constant

Task: make a checker that takes two template type arguments and check whether they are the same or not.

```
using true_type = integral_constant<bool, true>;
using false_type = integral_constant<bool, false>;

template <typename T, typename U>
struct is_same: false_type {};

template <typename T>
struct is_same<T, T>: true_type {};

std::cout << is_same<int, int>::value << std::endl;        // 1
std::cout << is_same<int, Matrix>::value << std::endl;     // 0
std::cout << is_same<Matrix, Matrix>::value << std::endl;  // 1
```

integral_constant

`Task: check that given type is reference?`

integral_constant

Task: check that given type is reference?

```
template <typename T>
struct is_reference: false_type {};
```

integral_constant

Task: check that given type is reference?

```
template <typename T>
struct is_reference: false_type {};

template <typename T>
struct is_reference<T&>: true_type {};

template <typename T>
struct is_reference<T&&>: true_type {};
```

integral_constant

Task: check that given type is reference?

```
template <typename T>
struct is_reference: false_type {};

template <typename T>
struct is_reference<T&>: true_type {};

template <typename T>
struct is_reference<T&&>: true_type {};


std::cout << is_reference<int&&>::value << std::endl;    // 1
std::cout << is_reference<char>::value << std::endl;     // 0
std::cout << is_reference<Matrix&>::value << std::endl;  // 1
```

integral_constant

Task: check that given type is integral (bool, char, int)?

integral_constant

Task: check that given type is integral (bool, char, int)?

```
template <typename T> struct is_integral: false_type {};

template <> struct is_integral<bool>: true_type {};
template <> struct is_integral<char>: true_type {};
template <> struct is_integral<int>: true_type {};
```

integral_constant

Task: check that given type is integral (bool, char, int)?

```
template <typename T> struct is_integral: false_type {};

template <> struct is_integral<bool>: true_type {};
template <> struct is_integral<char>: true_type {};
template <> struct is_integral<int>: true_type {};

std::cout << is_integral<int>::value << std::endl;      // 1
std::cout << is_integral<char>::value << std::endl;     // 1
std::cout << is_integral<Matrix&>::value << std::endl;  // 0
```

integral_constant

Task: check that given type is reference to integral (bool&, bool&&, char&, char&&, int&, int&&)?

integral_constant

Task: check that given type is reference to integral (bool&, bool&&, char&, char&&, int&, int&&)?

```
template <typename T, typename U>
struct and_: false_type {};

template <>
struct and_<true_type, true_type>: true_type {};
```

integral_constant

Task: check that given type is reference to integral (bool&, bool&&, char&, char&&, int&, int&&)?

```
template <typename T>
struct is_reference_to_integral: false_type {};
```

integral_constant

Task: check that given type is reference to integral (bool&, bool&&, char&, char&&, int&, int&&)?

```
template <typename T>
struct is_reference_to_integral: false_type {};

template <typename T>
struct is_reference_to_integral<T&>: and_<is_reference<T&>, is_integral<T>> {};

template <typename T>
struct is_reference_to_integral<T&&>: and_<is_reference<T&&>, is_integral<T>> {};
```

integral_constant

Task: check that given type is reference to integral (bool&, bool&&, char&, char&&, int&, int&&)?

```
template <typename T>
struct is_reference_to_integral: false_type {};

template <typename T>
struct is_reference_to_integral<T&>: and_<is_reference<T&>, is_integral<T>> {};

template <typename T>
struct is_reference_to_integral<T&&>: and_<is_reference<T&&>, is_integral<T>> {};


std::cout << is_reference_to_integral<int&>::value << std::endl;
```

integral_constant

```
Task: check that given type is reference to integral
(bool&, bool&&, char&, char&&, int&, int&&)?
```

```
template <typename T>
struct is_reference_to_integral: false_type {};

template <typename T>
struct is_reference_to_integral<T&>: and_<is_reference<T&>, is_integral<T>> {};

template <typename T>
struct is_reference_to_integral<T&&>: and_<is_reference<T&&>, is_integral<T>> {};


std::cout << is_reference_to_integral<int&>::value << std::endl; // 0
```

integral_constant

Task: check that given type is reference to integral (bool&, bool&&, char&, char&&, int&, int&&)?

```
template <typename T>
struct is_reference_to_integral: false_type {};

template <typename T>
struct is_reference_to_integral<T&>: and_<is_reference<T&>, is_integral<T>> {};

template <typename T>
struct is_reference_to_integral<T&&>: and_<is_reference<T&&>, is_integral<T>> {};


std::cout << is_reference_to_integral<int&>::value << std::endl; // 0… why?
```

integral_constant

Task: check that given type is reference to integral (bool&, bool&&, char&, char&&, int&, int&&)?

```
template <typename T, typename U>
struct and_: false_type {};

template <>
struct and_<true_type, true_type>: true_type {};
```

integral_constant

Task: check that given type is reference to integral (bool&, bool&&, char&, char&&, int&, int&&)?

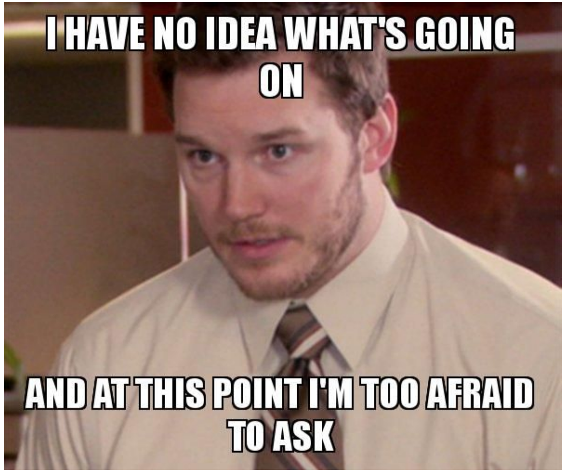
```
template <typename T, typename U>
struct and_: false_type {};

template <>
struct and_<true_type, true_type>: true_type {};
```

Yhese should be exactly true_type! Not derived from them!

integral_constant

Task: let's define a separate (wrapper) class for some instance of a class.

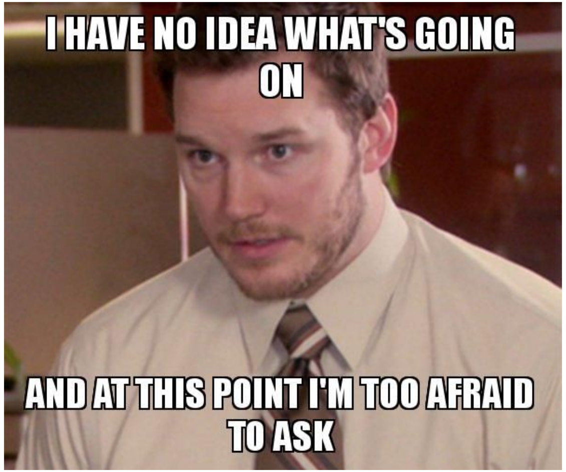
```
template <typename T, T v>
struct integral_constant {
   static const T value = v;
   typedef T value_type;
   typedef integral_constant type;
   operator value_type() const {
       return v;
   }
};

using ic42 = integral_constant<int, 42>;
int n = 13 * ic42{};
```



integral_constant

Task: let's define a separate (wrapper) class for some instance of a class.

```
template <typename T, T v>
struct integral_constant {
    static const T value = v;
    typedef T value_type;
    typedef integral_constant type;
    operator value_type() const {
        return v;
    }
};

using ic42 = integral_constant<int, 42>;
int n = 13 * ic42{};
```



integral_constant

Task: check that given type is reference to integral (bool&, bool&&, char&, char&&, int&, int&&)?

```
template <typename T>
struct is_reference_to_integral: false_type {};

template <typename T>
struct is_reference_to_integral<T&>: and_<is_reference<T&>, is_integral<T>> {};

template <typename T>
struct is_reference_to_integral<T&&>: and_<is_reference<T&&>, is_integral<T>> {};


std::cout << is_reference_to_integral<int&>::value << std::endl; // 0... why?
```

integral_constant

Task: check that given type is reference to integral (bool&, bool&&, char&, char&&, int&, int&&)?

```
template <typename T>
struct is_reference_to_integral: false_type {};

template <typename T>
struct is_reference_to_integral<T&>: and_<is_reference<T&>::type,
                                          is_integral<T>::type> {};


std::cout << is_reference_to_integral<int&>::value << std::endl; // 0... why?
```

integral_constant

Task: check that given type is reference to integral (bool&, bool&&, char&, char&&, int&, int&&)?

```
template <typename T>
struct is_reference_to_integral: false_type {};

template <typename T>
struct is_reference_to_integral<T&>: and_<typename is_reference<T&>::type,
                                          typename is_integral<T>::type> {};


std::cout << is_reference_to_integral<int&>::value << std::endl; // 1 !!!
```

integral_constant

Task: check that given type is integral (bool, char, int)?

```
template <typename T> struct is_integral: false_type {};

template <> struct is_integral<bool>: true_type {};
template <> struct is_integral<char>: true_type {};
template <> struct is_integral<int>: true_type {};
```

integral_constant

Task: check that given type is integral (bool, char, int)?

```
template <typename T> struct is_integral: false_type {};

template <> struct is_integral<bool>: true_type {};
template <> struct is_integral<char>: true_type {};
template <> struct is_integral<int>: true_type {};

template <typename T> using is_integral_t = typename is_integral<T>::type;
```

integral_constant

Task: check that given type is reference?

```
template <typename T>
struct is_reference: false_type {};

template <typename T>
struct is_reference<T&>: true_type {};

template <typename T>
struct is_reference<T&&>: true_type {};

template <typename T> using is_reference_t = typename is_reference<T>::type;
```

modifiers

```
template <typename T>
struct remove_reference {
   using type = T;
};
```

modifiers

```
template <typename T>
struct remove_reference {
   using type = T;
};

template <typename T>
struct remove_reference<T&> {
   using type = T;
};

template <typename T>
struct remove_reference<T&&> {
   using type = T;
};
```

modifiers

```
template <typename T>
struct remove_reference {
   using type = T;
};

template <typename T>
struct remove_reference<T&> {
   using type = T;
};

template <typename T>
struct remove_reference<T&&> {
   using type = T;
};

template <typename T>
using remove_reference_t = typename remove_reference<T>::type;
```

integral_constant

Task: check that given type is reference to integral (bool&, bool&&, char&, char&&, int&, int&&)?

```
template <typename T>
struct is_reference_to_integral: false_type {};

template <typename T>
struct is_reference_to_integral<T&>: and_<typename is_reference<T&>::type,
                                          typename is_integral<T>::type> {};


std::cout << is_reference_to_integral<int&>::value << std::endl; // 1 !!!
```

integral_constant

Task: check that given type is reference to integral (bool&, bool&&, char&, char&&, int&, int&&)?

```
template <typename T>
struct is_reference_to_integral: and_<is_reference_t<T>,
                                      is_integral_t<remove_reference_t<T>>> {};
```

integral_constant

Task: check that given type is reference to integral (bool&, bool&&, char&, char&&, int&, int&&)?

```
template <typename T>
struct is_reference_to_integral: and_<is_reference_t<T>,
                                      is_integral_t<remove_reference_t<T>>> {};
```

```
std::cout << is_reference_to_integral<int>::value << std::endl;        // 0
std::cout << is_reference_to_integral<int&>::value << std::endl;       // 1
std::cout << is_reference_to_integral<Matrix>::value << std::endl;     // 0
std::cout << is_reference_to_integral<Matrix&&>::value << std::endl;   // 0
```

checkers

`Note: std has all of these and much more.`

checkers

```
Note: std has all of these and much more.

Anything in C++ is among these 14 categories:

is_void
is_nullptr
is_integral, is_floating_point
is_array,
is_pointer,
is_lvalue_reference, is_right_value_reference,
is_member_object_pointer, is_member_function_pointer,
is_enum, is_union, is_class,
is_function
```

checkers

Note: std has all of these and much more.

There are a lot more checkers in std!

checkers

Note: std has all of these and much more.

There are a lot more checkers in std!

```
template<typename _Tp, typename _Up> struct is_convertable {...}
template<typename _Tp, typename _Up> struct is_base_of{...}
template<typename _Tp, typename _Up> struct is_same{...}
...

template<typename _Tp> struct is_constructable {...}
template<typename _Tp> struct is_destructable {...}
template<typename _Tp> struct is_copy_assignable {...}
...
```

https://en.cppreference.com/w/cpp/meta

Takeaways

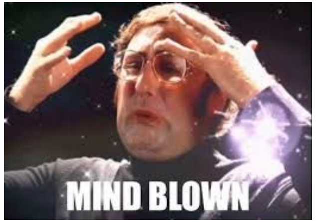
- 2 phase names lookup: be ready for ambiguous situations and don't be afraid of `template` and `typename` in unusual context

- SFINAE as a basic part of names lookup

- Checkers and modifiers as basic blocks that used SFINAE

Why to have all these stuff?

Templates: compile time execution (teaser)




```cpp
template <int N>
inline int fib() {
    return fib<N-1>() + fib<N-2>();
}

template <> inline int fib<1>() {...}
template <> inline int fib<2>() {...}

int main() {
    std::cout << fib<900>();
    return 0;
}
```

So, we will have an answer immediately, without any calculations in run-time.

```
int fib<1>():
        mov     eax, 1
        ret
int fib<2>():
        mov     eax, 1
        ret
main:
        sub     rsp, 8
        mov     esi, 372038192
        mov     edi, OFFSET FLAT:_ZSt4cout
        call    std::basic_ostream<char, std:
        xor     eax, eax
        add     rsp, 8
        ret
```

Templates: compile time execution (teaser #2)

This time we want to calculate the power of a number.

```cpp
template<int X, int Y> // X^Y = X*(X^(Y-1)) = X*(Pow(X, Y-1))
struct Pow {
    static const int result = X * Pow<X, Y - 1>::result;
};

template<int X>
struct Pow<X, 1> {
    static const int result = X;
};

int main() {
    std::cout << Pow<2, 30>::result;
    return 0;
}
```

```asm
main:
        push    rbp
        mov     rbp, rsp
        mov     esi, 1073741824
        mov     edi, OFFSET FLAT:_ZSt4cout
        call    std::basic_ostream<char, std::
        mov     eax, 0
        pop     rbp
        ret
```

https://godbolt.org/z/h9YMnd9v1

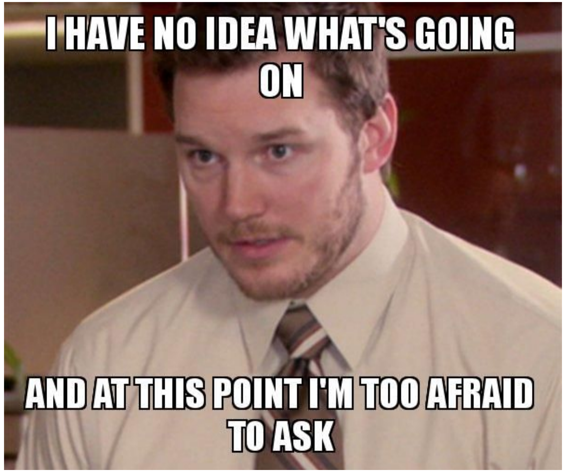
Templates: compile time execution

```cpp
template <int N>
inline int fib() {
    return fib<N-1>() + fib<N-2>();
}

template <> inline int fib<1>() {...}
template <> inline int fib<2>() {...}

int main() {
    std::cout << fib<900>();
    return 0;
}
```

integral_constant

Task: let's define a separate (wrapper) class for some instance of a class.

```
template <typename T, T v>
struct integral_constant {
    static const T value = v;
    typedef T value_type;
    typedef integral_constant type;
    operator value_type() const {
        return v;
    }
};

using ic42 = integral_constant<int, 42>;
int n = 13 * ic42{};
```



Templates: compile time execution

```
template <int N>
struct fib: std::integral_constant<int, fib<N - 1>{} + fib<N - 2>{}> {};
```

Templates: compile time execution

```
template <int N>
struct fib: std::integral_constant<int, fib<N - 1>{} + fib<N - 2>{}> {};

template <>
struct fib2<1>: std::integral_constant<int, 1> {};

template <>
struct fib2<2>: std::integral_constant<int, 1> {};
```

Templates: compile time execution

```
template <int N>
struct fib: std::integral_constant<int, fib<N - 1>{} + fib<N - 2>{}> {};

template <>
struct fib2<1>: std::integral_constant<int, 1> {};

template <>
struct fib2<2>: std::integral_constant<int, 1> {};

std::cout << fib2<46>{} << std::endl; // calculated in compile time
```

Templates: compile time execution

```
template <int N>
struct fib: std::integral_constant<int, fib<N - 1>{} + fib<N - 2>{}> {};

template <>
struct fib2<1>: std::integral_constant<int, 1> {};

template <>
struct fib2<2>: std::integral_constant<int, 1> {};

std::cout << fib2<46>{} << std::endl; // calculated in compile time
```

But what if we want to have something more complicated?
What about if/else?

Templates: compile time execution #3

```
Task: implement calculation of integer sqrt in compile time
```

Templates: compile time execution #3

Task: implement calculation of integer sqrt in compile time

```
int int_sqrt(int N, int lo, int hi) {
    int mid = (lo + hi + 1) / 2;
    if (lo == hi) {
        return lo;
    }
    if (N < mid*mid) {
        return int_sqrt(N, lo, mid - 1);
    } else {
        return int_sqrt(N, mid, hi);
    }
}
```

Basically a binary search!

Templates: compile time execution #3

Task: implement calculation of integer sqrt in compile time

```
int int_sqrt(int N, int lo, int hi) {
   int mid = (lo + hi + 1) / 2;
   if (lo == hi) {
       return lo;
   }
   if (N < mid*mid) {
       return int_sqrt(N, lo, mid - 1);
   } else {
       return int_sqrt(N, mid, hi);
   }
}
```

Basically a binary search!

Now let's move it to compile time.

Templates: compile time execution #3

Task: implement calculation of integer sqrt in compile time

```
int int_sqrt(int N, int lo, int hi) {
   int mid = (lo + hi + 1) / 2;
   if (lo == hi) {
       return lo;
   }
   if (N < mid*mid) {
       return int_sqrt(N, lo, mid - 1);
   } else {
       return int_sqrt(N, mid, hi);
   }
}
```

Basically a binary search!

Now let’s move it to compile time.

Conditional types

```
template <bool B, typename T, typename F>
struct conditional {
   using type = T;
};
```

Conditional types

```
template <bool B, typename T, typename F>
struct conditional {
   using type = T;
};

template <typename T, typename F>
struct conditional<false, T, F> {
   using type = F;
};
```

Conditional types

```
template <bool B, typename T, typename F>
struct conditional {
   using type = T;
};

template <typename T, typename F>
struct conditional<false, T, F> {
   using type = F;
};

template <bool B, typename T, typename F>
using conditional_t = typename conditional<B, T, F>::type;
```

Conditional types

Used, to check one type or another in compile time.

As usual, it is already in std.

```
template <bool B, typename T, typename F>
struct conditional {
    using type = T;
};

template <typename T, typename F>
struct conditional<false, T, F> {
    using type = F;
};

template <bool B, typename T, typename F>
using conditional_t = typename conditional<B, T, F>::type;
```

Templates: compile time execution #3

Task: implement calculation of integer sqrt in compile time

```
int int_sqrt(int N, int lo, int hi) {
    int mid = (lo + hi + 1) / 2;
    if (lo == hi) {
        return lo;
    }
    if (N < mid*mid) {
        return int_sqrt(N, lo, mid - 1);
    } else {
        return int_sqrt(N, mid, hi);
    }
}
```

Basically a binary search!

Now let's move it to compile time.

Templates: compile time execution #3

Task: implement calculation of integer sqrt in compile time

```
template <int N, int L = 1, int H = N, int mid = (L + H + 1) / 2>
struct ISqrt: std::integral_constant<int,
      std::conditional_t<(N < mid*mid),
                         ISqrt<N, L, mid - 1>,
                         ISqrt<N, mid, H>>{}> {};
```

Templates: compile time execution #3

Task: implement calculation of integer sqrt in compile time

```
template <int N, int L = 1, int H = N, int mid = (L + H + 1) / 2>
struct ISqrt: std::integral_constant<int,
```

Templates: compile time execution #3

Task: implement calculation of integer sqrt in compile time

```
template <int N, int L = 1, int H = N, int mid = (L + H + 1) / 2>
struct ISqrt: std::integral_constant<int,
      std::conditional_t<(N < mid*mid),
                         ISqrt<N, L, mid - 1>,
                         ISqrt<N, mid, H>>{}> {};


template <int N, int S>
struct ISqrt <N, S, S, S>: std::integral_constant<int, S> {};
```

Templates: compile time execution #3

Task: implement calculation of integer sqrt in compile time

```
template <int N, int L = 1, int H = N, int mid = (L + H + 1) / 2>
struct ISqrt: std::integral_constant<int,
      std::conditional_t<(N < mid*mid),
                         ISqrt<N, L, mid - 1>,
                         ISqrt<N, mid, H>>{}> {};


template <int N, int S>
struct ISqrt <N, S, S, S>: std::integral_constant<int, S> {};

std::cout << ISqrt<999>{} << std::endl; // 31
```

Why to have all these stuff?

1. `std::conditional` to choose between two types in compile time

Why to have all these stuff?

1. `std::conditional` to choose between two types in compile time

   In previous example we've chosen based on value, but all checkers can be used here to get some compile time logic based on type information.

enable_if

enable_if

enable_if allow you to directly remove some functions if condition is not met (with help of SFINAE).

enable_if

```
template <typename T,
          std::enable_if_t<std::is_trivially_copy_assignable<T>::value, int> = 0>
T* copy(T* dist, T* source) {
   std::cout << "It is trivially copy assignable" << std::endl;
   my_own_very_efficient_memcpy(dist, source, sizeof(T));
   return dist;
}
```

enable_if

```
template <typename T,
          std::enable_if_t<std::is_trivially_copy_assignable<T>::value, int> = 0>
T* copy(T* dist, T* source) {
   std::cout << "It is trivially copy assignable" << std::endl;
   my_own_very_efficient_memcpy(dist, source, sizeof(T));
   return dist;
}

template <typename T,
          std::enable_if_t<!std::is_trivially_copy_assignable<T>::value, int> = 0>

T* copy(T* dist, T* source) {
   std::cout << "It is NOT trivially copy assignable" << std::endl;
   *dist = *source;
   return dist;
}
```

```cpp
template <typename T,
          std::enable_if_t<std::is_trivially_copy_assignable<T>::value, int> = 0>
T* copy(T* dist, T* source) {
   std::cout << "It is trivially copy assignable" << std::endl;
   my_own_very_efficient_memcpy(dist, source, sizeof(T));
   return dist;
}

template <typename T,
          std::enable_if_t<!std::is_trivially_copy_assignable<T>::value, int> = 0>

T* copy(T* dist, T* source) {
   std::cout << "It is NOT trivially copy assignable" << std::endl;
   *dist = *source;
   return dist;
}

int* src = new int{}; *src = 10;
int* dst = new int{};
```

```
template <typename T,
          std::enable_if_t<std::is_trivially_copy_assignable<T>::value, int> = 0>
T* copy(T* dist, T* source) {
   std::cout << "It is trivially copy assignable" << std::endl;
   my_own_very_efficient_memcpy(dist, source, sizeof(T));
   return dist;
}

template <typename T,
          std::enable_if_t<!std::is_trivially_copy_assignable<T>::value, int> = 0>

T* copy(T* dist, T* source) {
   std::cout << "It is NOT trivially copy assignable" << std::endl;
   *dist = *source;
   return dist;
}

int* src = new int{}; *src = 10;
int* dst = new int{};

copy(dst, src);
```

```cpp
template <T = int, int dummy = 0>
T* copy(T* dist, T* source) {
   std::cout << "It is trivially copy assignable" << std::endl;
   my_own_very_efficient_memcpy(dist, source, sizeof(T));
   return dist;
}

template <T = int, /* something that can't be compiled */ = 0>
T* copy(T* dist, T* source) {
   std::cout << "It is NOT trivially copy assignable" << std::endl;
   *dist = *source;
   return dist;
}

int* src = new int{}; *src = 10;
int* dst = new int{};

copy(dst, src);
```

```cpp
template <T = int, int dummy = 0>
T* copy(T* dist, T* source) {
   std::cout << "It is trivially copy assignable" << std::endl;
   my_own_very_efficient_memcpy(dist, source, sizeof(T));
   return dist;
}

template <T = int, /* something that can’t be compiled */ = 0>
T* copy(T* dist, T* source) {
   std::cout << "It is NOT trivially copy assignable" << std::endl;
   *dist = *source;
   return dist;
}

int* src = new int{}; *src = 10;
int* dst = new int{};

copy(dst, src);
```

```cpp
template <T = int, int dummy = 0>
T* copy(T* dist, T* source) {
    std::cout << "It is trivially copy assignable" << std::endl;
    my_own_very_efficient_memcpy(dist, source, sizeof(T));
    return dist;
}

template <T = int, /* something that can't be compiled */ = 0>
T* copy(T* dist, T* source) {
    std::cout << "It is NOT trivially copy assignable" << std::endl;
    *dist = *source;
    return dist;
}

int* src = new int{}; *src = 10;
int* dst = new int{};

copy(dst, src);
```

Not a big deal,
SFINAE!

```
template <typename T,
          std::enable_if_t<std::is_trivially_copy_assignable<T>::value, int> = 0>
T* copy(T* dist, T* source) {
   std::cout << "It is trivially copy assignable" << std::endl;
   my_own_very_efficient_memcpy(dist, source, sizeof(T));
   return dist;
}

template <typename T,
          std::enable_if_t<!std::is_trivially_copy_assignable<T>::value, int> = 0>

T* copy(T* dist, T* source) {
   std::cout << "It is NOT trivially copy assignable" << std::endl;
   *dist = *source;
   return dist;
}

Vector<int>* source = new Vector<int>{16};
Vector<int>* dist = new Vector<int>{};

copy(dist, source);
```

```cpp
template <typename T, /* something that can’t be compiled */ = 0>
T* copy(T* dist, T* source) {
   std::cout << "It is trivially copy assignable" << std::endl;
   my_own_very_efficient_memcpy(dist, source, sizeof(T));
   return dist;
}

template <typename T, int dummy = 0>
T* copy(T* dist, T* source) {
   std::cout << "It is NOT trivially copy assignable" << std::endl;
   *dist = *source;
   return dist;
}

Vector<int>* source = new Vector<int>{16};
Vector<int>* dist = new Vector<int>{};

copy(dist, source);
```

```cpp
template <typename T, /* something that can’t be compiled */ = 0>
T* copy(T* dist, T* source) {
   std::cout << "It is trivially copy assignable" << std::endl;
   my_own_very_efficient_memcpy(dist, source, sizeof(T));
   return dist;
}
```

Not a big deal,
SFINAE!

```cpp
template <typename T, int dummy = 0>
T* copy(T* dist, T* source) {
   std::cout << "It is NOT trivially copy assignable" << std::endl;
   *dist = *source;
   return dist;
}

Vector<int>* source = new Vector<int>{16};
Vector<int>* dist = new Vector<int>{};

copy(dist, source);
```

Why to have all these stuff?

1. `std::conditional` to choose between two types in compile time

Why to have all these stuff?

1. std::conditional to choose between two types in compile time

2. Enable_if allow you to remove unsuitable functions or choose between different implementations in compile time!

Takeaways

- 2 phase names lookup: be ready for ambiguous situations and don't be afraid of `template` and `typename` in unusual context

- SFINAE as a basic part of names lookup

- Checkers and modifiers as basic blocks that used SFINAE

- Using those blocks you can build compiletime programs (Turing complete lang)

Not So Tiny Task №11 (0.5 point)

Calculate Nth prime number in compile-time! (without constexpr)