# Not So Tiny Task №9 (1 point)

Generalize data structure that you've implemented in NSTT #1, #3 and #4 with help of templates!

# Not So Tiny Task №10 (1 point)

Implement a mix-in to limit number of your class instances. Classes with such functionality should be successfully created only when there are less than specified number of instances, otherwise their construction should fail.
Use CRTP and non-type template argument (for the limit)

# System Programming with C++

## Templates

# System Programming with C++

## Templates

Eventually we will have standard catalogs of generic components with well-defined interfaces, with well-defined complexities. Programmers will stop programming at the micro level. You will never need to write a binary search routine again.

Alexander A. Stepanov, the author of STL

# Generic programming

```
int max(int x, int y) {
    return (x > y) ? x : y;
}
```

# Generic programming

```
int max(int x, int y) {
    return (x > y) ? x : y;
}
```

```
float max(float x, float y) {
    return (x > y) ? x : y;
}
```

# Generic programming

```
int max(int x, int y) {
    return (x > y) ? x : y;
}
```

```
float max(float x, float y) {
    return (x > y) ? x : y;
}
```

```
Matrix max(Matrix x, Matrix y) {
    return (x > y) ? x : y;
}
```

# Generic programming

What should Matrix have to make this method work?

```
int max(int x, int y) {
    return (x > y) ? x : y;
}
```

```
float max(float x, float y) {
    return (x > y) ? x : y;
}
```

```
Matrix max(Matrix x, Matrix y) {
    return (x > y) ? x : y;
}
```

# Generic programming

What should Matrix have to make this method work?

1. bool operator>(...)
2. copy constructor

```
int max(int x, int y) {
    return (x > y) ? x : y;
}
```

```
float max(float x, float y) {
    return (x > y) ? x : y;
}
```

```
Matrix max(Matrix x, Matrix y) {
    return (x > y) ? x : y;
}
```

# Generic programming

What should Matrix have to make this method work?

1. bool operator>(...)
2. copy constructor

```
int max(int x, int y) {
    return (x > y) ? x : y;
}
```

```
float max(float x, float y) {
    return (x > y) ? x : y;
}
```

```
Matrix max(Matrix x, Matrix y) {
    return (x > y) ? x : y;
}
```

How to fight with such copy-paste?

# Generic programming

What should Matrix have to make this method work?

1. bool operator>(...)
2. copy constructor

```
int max(int x, int y) {
    return (x > y) ? x : y;
}
```

```
float max(float x, float y) {
    return (x > y) ? x : y;
}
```

```
Matrix max(Matrix x, Matrix y) {
    return (x > y) ? x : y;
}
```

How to fight with such copy-paste?

Nothing to do with inheritance, as int/float/Matrix are not connected

# Generic programming

How other languages handle that?

# Generic programming

How other languages handle that?

```
def max(x, y) {
    return (x > y) ? x : y;
}
```

# Generic programming

How other languages handle that?

```
def max(x, y) {
    return (x > y) ? x : y;
}
```

Dynamically typed languages solve such problems easily: no types specified => you can pass anything here

# Generic programming

How other languages handle that?

```
def max(x, y) {
    return (x > y) ? x : y;
}
```


Dynamically typed languages solve such problems easily: no types specified => you can pass anything here

But there are consequences:

# Generic programming

How other languages handle that?

```
def max(x, y) {
    return (x > y) ? x : y;
}
```


Dynamically typed languages solve such problems easily: no types specified => you can pass anything here

But there are consequences:

1) Passing unsuitable types will cause RT exception

# Generic programming

How other languages handle that?

```
def max(x, y) {
    return (x > y) ? x : y;
}
```


Dynamically typed languages solve such problems easily: no types specified => you can pass anything here

But there are consequences:

1) Passing unsuitable types will cause RT exception

2) Would be nice to find a problem before execution

# Generic programming

How other languages handle that?

```
def max(x, y) {
    return (x > y) ? x : y;
}
```


Dynamically typed languages solve such problems easily: no types specified => you can pass anything here

But there are consequences:

3) Performance costs (to find a method and calls will be virtual of course)

# Generic programming

How other languages handle that?

```java
public static <T extends Comparable<T>> T max(T a, T b) {
    if (a == null) {
        if (b == null) return a;
        else return b;
    }
    if (b == null) return a;
    return a.compareTo(b) > 0 ? a : b;
}
```

# Generic programming

In Java we still try to solve it with hierarchies, because this is a way to specify requirements to this generic T argument.

How other languages handle that?

```
public static <T extends Comparable<T>> T max(T a, T b) {
    if (a == null) {
        if (b == null) return a;
        else return b;
    }
    if (b == null) return a;
    return a.compareTo(b) > 0 ? a : b;
}
```

# Generic programming

In Java we still try to solve it with hierarchies, because this is a way to specify requirements to this generic T argument.

How other languages handle that?

```
public static <T extends Comparable<T>> T max(T a, T b) {
    if (a == null) {
        if (b == null) return a;
        else return b;
    }
    if (b == null) return a;
    return a.compareTo(b) > 0 ? a : b;
}
```

Problems:

1. Doesn’t work with primitives without boxing

# Generic programming

In Java we still try to solve it with hierarchies, because this is a way to specify requirements to this generic T argument.

How other languages handle that?

```
public static <T extends Comparable<T>> T max(T a, T b) {
    if (a == null) {
        if (b == null) return a;
        else return b;
    }
    if (b == null) return a;
    return a.compareTo(b) > 0 ? a : b;
}
```

Problems:

1. Doesn’t work with primitives without boxing
2. Type information is lost because of implementation (performance drop)

# Generic programming

In Java we still try to solve it with hierarchies, because this is a way to specify requirements to this generic T argument.

How other languages handle that?

```
public static <T extends Comparable<T>> T max(T a, T b) {
    if (a == null) {
        if (b == null) return a;
        else return b;
    }
    if (b == null) return a;
    return a.compareTo(b) > 0 ? a : b;
}
```

Problems:

1. Doesn't work with primitives without boxing
2. Type information is lost because of implementation (performance drop)
3. Requirements via inheritance is not always good

# Generic programming in C++

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}
```

# Generic programming in C++

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}
```

both typename and class can be used here (and there discussions about it), I will use typename in lectures.

# Generic programming in C++

```cpp
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}
```

T - formal template parameter

# Generic programming in C++

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}

-----------------------------

int a, b, c;
...
a = max<int>(b, c);
```

T - formal template parameter

int - actual parameter type in this instantiation.

# Generic programming in C++

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}
```

T - formal template parameter

```
---------------------------

float a, b, c;
...
a = max<float>(b, c);
```

float - actual parameter type in this instantiation.

# Generic programming in C++

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}
```

T - formal template parameter

----------------------------

```
Matrix a, b, c;
...
a = max<Matrix>(b, c);
```

Matrix - actual parameter type in this instantiation. Will it compile?

# Generic programming in C++

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}
```

T - formal template parameter

```
----------------------------

Matrix a, b, c;
...
a = max<Matrix>(b, c);
```

Matrix - actual parameter type in this instantiation. Will it compile? It depends on Matrix!

# Generic programming in C++

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}
```

T - formal template parameter

```
In instantiation of 'T max(T, T) [with T = Matrix]':
error: no match for 'operator>' (operand types are
'Matrix' and 'Matrix')
    return (a > b) ? a : b;
           ~~~^~~~
```

```
---------------------------

Matrix a, b, c;
...
a = max<Matrix>(b, c);
```

Matrix - actual parameter type in this instantiation. Will it compile? It depends on Matrix!

# Generic programming in C++

The code of the method max defines requirements for the type that will be substituted

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}
```

T - formal template parameter

```
In instantiation of 'T max(T, T) [with T = Matrix]':
error: no match for 'operator>' (operand types are
'Matrix' and 'Matrix')
    return (a > b) ? a : b;
           ~~~^~~~
```

```
---------------------------

Matrix a, b, c;
...
a = max<Matrix>(b, c);
```

Matrix - actual parameter type in this instantiation. Will it compile? It depends on Matrix!

# Generic programming in C++

The code of the method max defines requirements for the type that will be substituted

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}

---------------------------

Matrix a, b, c;
...
a = max<Matrix>(b, c);
```

T - formal template parameter

```
class Matrix {

   Matrix(const Matrix& other) {
       ...
   }

   bool operator>(const Matrix& other) {
       ...
   }

};
```

# Generic programming in C++

The code of the method max defines requirements for the type that will be substituted

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}
```

---

```
Matrix a, b, c;
...
a = max<Matrix>(b, c);
```

T - formal template parameter

```
class Matrix {

    Matrix(const Matrix& other) {
        ...
    }

    bool operator>(const Matrix& other) {
        ...
    }

};
```

Now this will indeed compile.

# Templates: how does it work?

```cpp
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}
```

```cpp
int a, b = 3, c= 10;
a = max<int>(b, c);

Matrix k, l, m;
k = max<Matrix>(l, m);
```

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}
```

Compiler finds set of all actual types:

T = {int, Matrix}

```
int a, b = 3, c= 10;
a = max<int>(b, c);

Matrix k, l, m;
k = max<Matrix>(l, m);
```

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}
```

Compiler finds set of all actual types:

T = {int, Matrix}

Then generates versions of method max (instantiate)

```
int max_1(int x, int y) {
    return (x > y) ? x : y;
}
```

```
Matrix max_2(Matrix x, Matrix y) {
    return (x > y) ? x : y;
}
```

```
int a, b = 3, c= 10;
a = max<int>(b, c);

Matrix k, l, m;
k = max<Matrix>(l, m);
```

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}
```

Compiler finds set of all actual types:

```
T = {int, Matrix}
```

Then generates versions of method max (instantiate). And updates calls!

```
int max_1(int x, int y) {
    return (x > y) ? x : y;
}
```

```
Matrix max_2(Matrix x, Matrix y) {
    return (x > y) ? x : y;
}
```

```
int a, b = 3, c= 10;
a = max_1(b, c);
```

```
Matrix k, l, m;
k = max_2(l, m);
```

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}
```

```
int max_1(int x, int y) {
    return (x > y) ? x : y;
}
```

```
Matrix max_2(Matrix x, Matrix y) {
    return (x > y) ? x : y;
}
```

Compiler finds set of all actual types:

```
T = {int, Matrix}
```

Then generates versions of method max (instantiate). And updates calls! This process is called monomorphization.

```
int a, b = 3, c= 10;
a = max_1(b, c);
```

```
Matrix k, l, m;
k = max_2(l, m);
```

```
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16
    mov     DWORD PTR [rbp-4], 3
    mov     DWORD PTR [rbp-8], 10
    mov     edx, DWORD PTR [rbp-8]
    mov     eax, DWORD PTR [rbp-4]
    mov     esi, edx
    mov     edi, eax
    call    int max<int>(int, int)
    mov     DWORD PTR [rbp-12], eax
    call    Matrix max<Matrix>(Matrix, Matrix)
    mov     eax, DWORD PTR [rbp-12]
    leave
    ret
```

```
int max<int>(int, int):
    push    rbp
    mov     rbp, rsp
    mov     DWORD PTR [rbp-4], edi
    mov     DWORD PTR [rbp-8], esi
    mov     eax, DWORD PTR [rbp-4]
    cmp     eax, DWORD PTR [rbp-8]
    jle     .L6
    mov     eax, DWORD PTR [rbp-4]
    jmp     .L8
.L6:
    mov     eax, DWORD PTR [rbp-8]
.L8:
    pop     rbp
    ret
```

```
Matrix max<Matrix>(Matrix, Matrix):
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16
    lea     rdx, [rbp-2]
    lea     rax, [rbp-1]
    mov     rsi, rdx
    mov     rdi, rax
    call    Matrix::operator>(Matrix&)
    test    al, al
    nop
    leave
    ret
```

https://godbolt.org/z/KMMEnvT7M

```
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16
    mov     DWORD PTR [rbp-4], 3
    mov     DWORD PTR [rbp-8], 10
    mov     edx, DWORD PTR [rbp-8]
    mov     eax, DWORD PTR [rbp-4]
    mov     esi, edx
    mov     edi, eax
    call    int max<int>(int, int)
    mov     DWORD PTR [rbp-12], eax
    call    Matrix max<Matrix>(Matrix, Matrix)
    mov     eax, DWORD PTR [rbp-12]
    leave
    ret
```

```
int max<int>(int, int):
    push    rbp
    mov     rbp, rsp
    mov     DWORD PTR [rbp-4], edi
    mov     DWORD PTR [rbp-8], esi
    mov     eax, DWORD PTR [rbp-4]
    cmp     eax, DWORD PTR [rbp-8]
    jle     .L6
    mov     eax, DWORD PTR [rbp-4]
    jmp     .L8
.L6:
    mov     eax, DWORD PTR [rbp-8]
.L8:
    pop     rbp
    ret
```

```
Matrix max<Matrix>(Matrix, Matrix):
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16
    lea     rdx, [rbp-2]
    lea     rax, [rbp-1]
    mov     rsi, rdx
    mov     rdi, rax
    call    Matrix::operator>(Matrix&)
    test    al, al
    nop
    leave
    ret
```

https://godbolt.org/z/KMMEnvT7M

```
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16
    mov     DWORD PTR [rbp-4], 3
    mov     DWORD PTR [rbp-8], 10
    mov     edx, DWORD PTR [rbp-8]
    mov     eax, DWORD PTR [rbp-4]
    mov     esi, edx
    mov     edi, eax
    call    int max<int>(int, int)
    mov     DWORD PTR [rbp-12], eax
    call    Matrix max<Matrix>(Matrix, Matrix)
    mov     eax, DWORD PTR [rbp-12]
    leave
    ret
```

```
int max<int>(int, int):
    push    rbp
    mov     rbp, rsp
    mov     DWORD PTR [rbp-4], edi
    mov     DWORD PTR [rbp-8], esi
    mov     eax, DWORD PTR [rbp-4]
    cmp     eax, DWORD PTR [rbp-8]
    jle     .L6
    mov     eax, DWORD PTR [rbp-4]
    jmp     .L8
.L6:
    mov     eax, DWORD PTR [rbp-8]
.L8:
    pop     rbp
    ret
```

```
Matrix max<Matrix>(Matrix, Matrix):
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16
    lea     rdx, [rbp-2]
    lea     rax, [rbp-1]
    mov     rsi, rdx
    mov     rdi, rax
    call    Matrix::operator>(Matrix&)
    test    al, al
    nop
    leave
    ret
```

https://godbolt.org/z/KMMEnvT7M

# Monomorphization: pros and cons

# Monomorphization: pros and cons

Pros:

1. Well, this is basically zero-cost abstraction.

# Monomorphization: pros and cons

Pros:

1. Well, this is basically zero-cost abstraction. It means it will give you no performance drop during execution of the application. It will work fast!

# Monomorphization: pros and cons

Pros:

1. Well, this is basically zero-cost abstraction. It means it will give you no performance drop during execution of the application. It will work fast!

2. Type-safety is still here: no sudden exceptions during execution, compiler will check types!

# Monomorphization: pros and cons

Pros:

1. Well, this is basically zero-cost abstraction. It means it will give you no performance drop during execution of the application. It will work fast!

2. Type-safety is still here: no sudden exceptions during execution, compiler will check types!

Cons?

# Monomorphization: pros and cons

Pros:

1. Well, this is basically zero-cost abstraction. It means it will give you no performance drop during execution of the application. It will work fast!

2. Type-safety is still here: no sudden exceptions during execution, compiler will check types!

Cons:

1. Compilation time/size of binary.

# Monomorphization: pros and cons

Pros:

1. Well, this is basically zero-cost abstraction. It means it will give you no performance drop during execution of the application. It will work fast!

2. Type-safety is still here: no sudden exceptions during execution, compiler will check types!

Cons:

1. Compilation time/size of binary.
2. Errors in templates can be a mess

```
const string&)>&>; T2 = jinja2::EmptyValue]’
/home/travis/build/flexferrum/Jinja2Cpp/thirdparty/nonstd/variant-light/include/nonstd/variant.hpp:1597:17:   required from ‘static R nonstd::variants::detail::VisitorApplicatorImpl<R, VT>::apply(const Visitor&, const T&) [with Visitor = nonstd::variants::detail::TypedVisitorUnwrapper<2ul, jinja2::Value, jinja2::detail::UCInvoker<const UserCallableTest_SimpleUserCallableWithParams2_Test::TestBody()::<lambda(const string&, const string&)>&>, jinja2::EmptyValue>; T = bool; R = jinja2::Value; VT = bool]’
/home/travis/build/flexferrum/Jinja2Cpp/thirdparty/nonstd/variant-light/include/nonstd/variant.hpp:1807:59:   required from ‘static R nonstd::variants::detail::VisitorApplicator<R>::apply_visitor(const Visitor&, const V1&) [with long unsigned int Idx = 1ul; Visitor = nonstd::variants::detail::TypedVisitorUnwrapper<2ul, jinja2::Value, jinja2::detail::UCInvoker<const UserCallableTest_SimpleUserCallableWithParams2_Test::TestBody()::<lambda(const string&, const string&)>&>, jinja2::EmptyValue>; V1 = nonstd::variants::variant<jinja2::EmptyValue, bool, std::basic_string<char>, std::basic_string<wchar_t>, long int, double, nonstd::vptr::value_ptr<std::vector<jinja2::Value>, nonstd::vptr::detail::default_clone<std::vector<jinja2::Value> >, std::default_delete<std::vector<jinja2::Value> > >, nonstd::vptr::value_ptr<std::unordered_map<std::basic_string<char>, jinja2::Value>, nonstd::vptr::detail::default_clone<std::unordered_map<std::basic_string<char>, jinja2::Value> >, std::default_delete<std::unordered_map<std::basic_string<char>, jinja2::Value> > >, jinja2::GenericList, jinja2::GenericMap, nonstd::vptr::value_ptr<jinja2::UserCallable, nonstd::vptr::detail::default_clone<jinja2::UserCallable>, std::default_delete<jinja2::UserCallable> > >; R = jinja2::Value]’
/home/travis/build/flexferrum/Jinja2Cpp/thirdparty/nonstd/variant-light/include/nonstd/variant.hpp:1778:44:   required from ‘static R nonstd::variants::detail::VisitorApplicator<R>::apply(const Visitor&, const V1&) [with Visitor = nonstd::variants::detail::TypedVisitorUnwrapper<2ul, jinja2::Value, jinja2::detail::UCInvoker<const UserCallableTest_SimpleUserCallableWithParams2_Test::TestBody()::<lambda(const string&, const string&)>&>, jinja2::EmptyValue>; V1 = nonstd::variants::variant<jinja2::EmptyValue, bool, std::basic_string<char>, std::basic_string<wchar_t>, long int, double, nonstd::vptr::value_ptr<std::vector<jinja2::Value>, nonstd::vptr::detail::default_clone<std::vector<jinja2::Value> >, std::default_delete<std::vector<jinja2::Value> > >, nonstd::vptr::value_ptr<std::unordered_map<std::basic_string<char>, jinja2::Value>, nonstd::vptr::detail::default_clone<std::unordered_map<std::basic_string<char>, jinja2::Value> >, std::default_delete<std::unordered_map<std::basic_string<char>, jinja2::Value> > >, jinja2::GenericList, jinja2::GenericMap, nonstd::vptr::value_ptr<jinja2::UserCallable, nonstd::vptr::detail::default_clone<jinja2::UserCallable>, std::default_delete<jinja2::UserCallable> > >; R = jinja2::Value]’
/home/travis/build/flexferrum/Jinja2Cpp/thirdparty/nonstd/variant-light/include/nonstd/variant.hpp:1735:43:   [ skipping 5 instantiation contexts, use -ftemplate-backtrace-limit=0 to disable ]
/home/travis/build/flexferrum/Jinja2Cpp/thirdparty/nonstd/variant-light/include/nonstd/variant.hpp:1871:45:   required from ‘typename nonstd::variants::detail::VisitorImpl<sizeof... (V), Visitor, V ...>::result_type nonstd::variants::visit(const Visitor&, const V& ...) [with Visitor = jinja2::detail::UCInvoker<const UserCallableTest_SimpleUserCallableWithParams2_Test::TestBody()::<lambda(const string&, const string&)>&>; V = {nonstd::variants::variant<jinja2::EmptyValue, bool, std::basic_string<char, std::char_traits<char>, std::allocator<char> >, std::basic_string<wchar_t, std::char_traits<wchar_t>, std::allocator<wchar_t> >, long int, double, nonstd::vptr::value_ptr<std::vector<jinja2::Value, std::allocator<jinja2::Value> >, nonstd::vptr::detail::default_clone<std::vector<jinja2::Value, std::allocator<jinja2::Value> > >, std::default_delete<std::vector<jinja2::Value, std::allocator<jinja2::Value> > > >, nonstd::vptr::value_ptr<std::unordered_map<std::basic_string<char, std::char_traits<char>, std::allocator<char> >, jinja2::Value, std::hash<std::basic_string<char, std::char_traits<char>, std::allocator<char> > >, std::equal_to<std::basic_string<char, std::char_traits<char>, std::allocator<char> > >, std::allocator<std::pair<const std::basic_string<char, std::char_traits<char>, std::allocator<char> >, jinja2::Value> > >, nonstd::vptr::detail::default_clone<std::unordered_map<std::basic_string<char, std::char_traits<char>, std::allocator<char> >, jinja2::Value, std::hash<std::basic_string<char, std::char_traits<char>, std::allocator<char> > >, std::equal_to<std::basic_string<char, std::char_traits<char>, std::allocator<char> > >, std::allocator<std::pair<const std::basic_string<char, std::char_traits<char>, std::allocator<char> >, jinja2::Value> > > >, std::default_delete<std::unordered_map<std::basic_string<char, std::char_traits<char>, std::allocator<char> >, jinja2::Value, std::hash<std::basic_string<char, std::char_traits<char>, std::allocator<char> > >, std::equal_to<std::basic_string<char, std::char_traits<char>, std::allocator<char> > >, std::allocator<std::pair<const std::basic_string<char, std::char_traits<char>, std::allocator<char> >, jinja2::Value> > > > >, jinja2::GenericList, jinja2::GenericMap, nonstd::vptr::value_ptr<jinja2::UserCallable, nonstd::vptr::detail::default_clone<jinja2::UserCallable>, std::default_delete<jinja2::UserCallable> >, nonstd::variants::detail::TX<nonstd::variants::detail::S11>, nonstd::variants::detail::TX<nonstd::variants::detail::S12>, nonstd::variants::detail::TX<nonstd::variants::detail::S13>, nonstd::variants::detail::TX<nonstd::variants::detail::S14>, nonstd::variants::detail::TX<nonstd::variants::detail::S15> >, nonstd::variants::variant<jinja2::EmptyValue, bool, std::basic_string<char, std::char_traits<char>, std::allocator<char> >, std::basic_string<wchar_t, std::char_traits<wchar_t>, std::allocator<wchar_t> >, long int, double, nonstd::vptr::value_ptr<std::vector<jinja2::Value, std::allocator<jinja2::Value> >, nonstd::vptr::detail::default_clone<std::vector<jinja2::Value, std::allocator<jinja2::Value> > >, std::default_delete<std::vector<jinja2::Value, std::allocator<jinja2::Value> > > >, nonstd::vptr::value_ptr<std::unordered_map<std::basic_string<char, std::char_traits<char>, std::allocator<char> >, jinja2::Value, std::hash<std::basic_string<char, std::char_traits<char>, std::allocator<char> > >, std::equal_to<std::basic_string<char, std::char_traits<char>, std::allocator<char> > >, std::allocator<std::pair<const std::basic_string<char, std::char_traits<char>, std::allocator<char> >, jinja2::Value> > >, nonstd::vptr::detail::default_clone<std::unordered_map<std::basic_string<char, std::char_traits<char>, std::allocator<char> >, jinja2::Value, std::hash<std::basic_string<char, std::char_traits<char>, std::allocator<char> > >, std::equal_to<std::basic_string<char, std::char_traits<char>, std::allocator<char> > >, std::allocator<std::pair<const std::basic_string<char, std::char_traits<char>, std::allocator<char> >, jinja2::Value> > > >, std::default_delete<std::unordered_map<std::basic_string<char, std::char_traits<char>, std::allocator<char> >, jinja2::Value,
```

# Specialization

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}
```

# Specialization

```cpp
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}

template <>
Matrix max(Matrix x, Matrix y) {
    return (x[0][0] > y[0][0]) ? x : y;
}
```

# Specialization

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}

template <>
Matrix max(Matrix x, Matrix y) {
    return (x[0][0] > y[0][0]) ? x : y;
}
```

Explicitly specialize a template with some concrete type

# Specialization

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}

template <>
Matrix max(Matrix x, Matrix y) {
    return (x[0][0] > y[0][0]) ? x : y;
}

Matrix c = max<Matrix>(a, b);
```

Explicitly specialize a template with some concrete type

# Specialization

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}

template <>
Matrix max(Matrix x, Matrix y) {
    return (x[0][0] > y[0][0]) ? x : y;
}

Matrix c = max<Matrix>(a, b);
```

Explicitly specialize a template with some concrete type

Specialized version will be called

# Specialization

```cpp
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}
```

Primary template definition

```cpp
template <>
Matrix max(Matrix x, Matrix y) {
    return (x[0][0] > y[0][0]) ? x : y;
}
```

Explicitly specialize a template with some concrete type

```cpp
Matrix c = max<Matrix>(a, b);
```

Specialized version will be called

# Specialization

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}

Matrix c = max<Matrix>(a, b);

template <>
Matrix max(Matrix x, Matrix y) {
    return (x[0][0] > y[0][0]) ? x : y;
}
```

```cpp
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}

void foo() {
   Matrix a, b;
   Matrix c = max<Matrix>(a, b);
}

template <>
Matrix max(Matrix x, Matrix y) {
    return (x[0][0] > y[0][0]) ? x : y;
}
```

```cpp
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}

void foo() {
   Matrix a, b;
   Matrix c = max<Matrix>(a, b);
}

template <>
Matrix max(Matrix x, Matrix y) {
    return (x[0][0] > y[0][0]) ? x : y;
}
```

```
error: specialization of 'T max(T, T)
[with T = Matrix]' after instantiation
  749 | Matrix max(Matrix x, Matrix y) {
                                    ^
```

Order of declaration matters!

```cpp
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}

void foo() {
   Matrix a, b;
   Matrix c = max<Matrix>(a, b);
}

template <>
Matrix max(Matrix x, Matrix y) {
    return (x[0][0] > y[0][0]) ? x : y;
}
```

```
error: specialization of 'T max(T, T)
[with T = Matrix]' after instantiation
  749 | Matrix max(Matrix x, Matrix y) {
                                    ^
```

```cpp
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}

void foo() {
   Matrix a, b;
   Matrix c = max<Matrix>(a, b);
}

template <>
Matrix max(Matrix x, Matrix y) {
    return (x[0][0] > y[0][0]) ? x : y;
}
```

Order of declaration matters!
If instantiation comes first, before specialization => there will be a compilation error.

```
error: specialization of 'T max(T, T)
[with T = Matrix]' after instantiation
  749 | Matrix max(Matrix x, Matrix y) {
                                    ^
```

```cpp
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}

template <>
Matrix max(Matrix x, Matrix y) = delete;
```

```cpp
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}

template <>
Matrix max(Matrix x, Matrix y) = delete;
```

You can prohibit anyone to specialize your template for some concrete classes.

```cpp
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}

template <>
Matrix max(Matrix x, Matrix y) = delete;
```

You can prohibit anyone to specialize your template for some concrete classes. But again: this specialization should be placed before any instantiation!

# Template classes

# Template classes

```
class Vector {
   size_t size_;
   size_t cap_;
   int* data_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        cap_ = initial_capacity;
        data_ = new int[cap_];
    }

    int& operator[](size_t index) {
       return data_[index];
    }
};
```

Why the hell elements are ints?

# Template classes

```
class Vector {
   size_t size_;
   size_t cap_;
   int* data_;
public:
    Vector(size_t initial_capacity) {
         size_ = 0;
         cap_ = initial_capacity;
         data_ = new int[cap_];
    }

    int& operator[](size_t index) {
       return data_[index];
    }
};
```

Why the hell elements are ints?

# Template classes

```
template<typename T>
class Vector {
   size_t size_;
   size_t cap_;
   int* data_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        cap_ = initial_capacity;
        data_ = new int[cap_];
    }

    int& operator[](size_t index) {
       return data_[index];
    }
};
```

Why the hell elements are ints?

# Template classes

```
template<typename T>
class Vector {
   size_t size_;
   size_t cap_;
   T* data_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        cap_ = initial_capacity;
        data_ = new T[cap_];
    }

    T& operator[](size_t index) {
       return data_[index];
    }
};
```

Why the hell elements are ints?

# Template classes

```
template<typename T>
class Vector {
   size_t size_;
   size_t cap_;
   T* data_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        cap_ = initial_capacity;
        data_ = new T[cap_];
    }

    T& operator[](size_t index) {
       return data_[index];
    }
};
```

```
Vector<int>    v1{16};
Vector<float>  v2{8};
Vector<Matrix> v3{};
```

Why the hell elements are ints?

# Template classes

```
template<typename T>
class Vector {
   size_t size_;
   size_t cap_;
   T* data_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        cap_ = initial_capacity;
        data_ = new T[cap_];
    }

    T& operator[](size_t index) {
       return data_[index];
    }
};
```

```
Vector<int>    v1{16};
Vector<float>  v2{8};
Vector<Matrix> v3{};
```

Implemented (almost) the same.

Why the hell elements are ints?

# Template classes

```
template<typename T>
class Vector {
   size_t size_;
   size_t cap_;
   T* data_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        cap_ = initial_capacity;
        data_ = new T[cap_];
    }

    T& operator[](size_t index) {
       return data_[index];
    }
};
```

```
Vector<int>    v1{16};
Vector<float>  v2{8};
Vector<Matrix> v3{};
```

Implemented (almost) the same.

Compiler again finds all possible actual template parameters.

Why the hell elements are ints?

# Template classes

```
template<typename T>
class Vector {
   size_t size_;
   size_t cap_;
   T* data_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        cap_ = initial_capacity;
        data_ = new T[cap_];
    }

    T& operator[](size_t index) {
       return data_[index];
    }
};
```

```
Vector<int>    v1{16};
Vector<float>  v2{8};
Vector<Matrix> v3{};
```

Implemented (almost) the same.

Compiler again finds all possible actual template parameters.

Ang generates… well, some part of corresponding classes.

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

# Template classes

```
template<typename T>
class Vector {
   size_t size_;
   size_t cap_;
   T* data_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        cap_ = initial_capacity;
        data_ = new T[cap_];
    }

    T& operator[](size_t index) {
       return data_[index]->size;
    }
};
```

```
Vector<int>    v1{16};
Vector<float>  v2{8};
Vector<Matrix> v3{};
```

# Template classes

```
template<typename T>
class Vector {
   size_t size_;
   size_t cap_;
   T* data_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        cap_ = initial_capacity;
        data_ = new T[cap_];
    }

    T& operator[](size_t index) {
       return data_[index]->size;
    }
};
```

```
Vector<int>    v1{16};
Vector<float>  v2{8};
Vector<Matrix> v3{};
```

```
Build finished OK
```

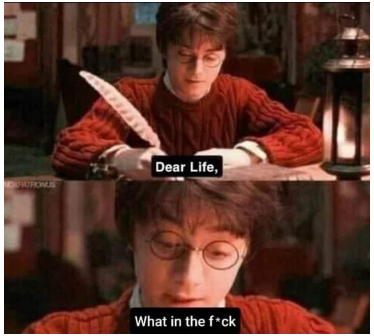
# Template classes

```
template<typename T>
class Vector {
   size_t size_;
   size_t cap_;
   T* data_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        cap_ = initial_capacity;
        data_ = new T[cap_];
    }

    T& operator[](size_t index) {
       return data_[index]->size;
    }
};
```

```
Vector<int>    v1{16};
Vector<float>  v2{8};
Vector<Matrix> v3{};
```

```
Build finished OK
```

Why???
What is data_[index]->size, if data_ is int*?



# Template classes

```cpp
template<typename T>
class Vector {
   size_t size_;
   size_t cap_;
   T* data_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        cap_ = initial_capacity;
        data_ = new T[cap_];
    }

    T& operator[](size_t index) {
       return data_[index]->size;
    }
};
```

```cpp
Vector<int>    v1{16};
Vector<float>  v2{8};
Vector<Matrix> v3{};
```

```
Build finished OK
```

Instantiation of functions inside template classes is made lazily.

# Template classes

```
template<typename T>
class Vector {
   size_t size_;
   size_t cap_;
   T* data_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        cap_ = initial_capacity;
        data_ = new T[cap_];
    }

    T& operator[](size_t index) {
       return data_[index]->size;
    }
};
```

```
Vector<int>    v1{16};
Vector<float>  v2{8};
Vector<Matrix> v3{};
```

Build finished OK

Instantiation of functions inside template classes is made lazily.

If there are no uses of some function, it will not be generated during monomorphization.

# Template classes

```
template<typename T>
class Vector {
   size_t size_;
   size_t cap_;
   T* data_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        cap_ = initial_capacity;
        data_ = new T[cap_];
    }

    T& operator[](size_t index) {
       return data_[index]->size;
    }
};
```

```
Vector<int> v1{16};
int x = v1[10];
```

# Template classes

```
template<typename T>
class Vector {
   size_t size_;
   size_t cap_;
   T* data_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        cap_ = initial_capacity;
        data_ = new T[cap_];
    }

    T& operator[](size_t index) {
       return data_[index]->size;
    }
};
```

```
 Vector<int> v1{16};
 int x = v1[10];
```

```
In instantiation of
'T& Vector<T>::operator[](size_t)
[with T = int;
      size_t = long long unsigned int]':
error: base operand of '->' is not a pointer
   86 |         return data_[index]->size;
      |                             ^~~~~
```

# Template classes

```cpp
template<typename T>
class Vector {
   size_t size_;
   size_t cap_;
   T* data_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        cap_ = initial_capacity;
        data_ = new T[cap_];
    }

    T& operator[](size_t index) {
       return data_[index]->size;
    }
};
```

```cpp
Vector<int> v1{16};
int x = v1[10];
```

```
In instantiation of
'T& Vector<T>::operator[](size_t)
[with T = int;
      size_t = long long unsigned int]':
error: base operand of '->' is not a pointer
   86 |         return data_[index]->size;
      |                            ^~~~~
```

We have a usage, so, compiler tried to generate operator[] (and failed)

# Template classes

Why instantiation of functions in template classes is lazy?

# Template classes

Why instantiation of functions in template classes is lazy?

1. To reduce compilation time,
2. To reduce size of binary,

# Template classes

Why instantiation of functions in template classes is lazy?

1. To reduce compilation time,
2. To reduce size of binary,

It is a bit country-intuitive, but uncontrolled generation of code can be a problem (imagine that you have many template arguments)

# Template classes

Why instantiation of functions in template classes is lazy?

1. To reduce compilation time,
2. To reduce size of binary,

3. You can indeed use only some parts of the template class for actual template parameter,

# Template classes

Why instantiation of functions in template classes is lazy?

1. To reduce compilation time,
2. To reduce size of binary,

3. You can indeed use only some parts of the template class for actual template parameter,

4. It will break some meta-programming stuff (will discuss later)

# Template classes

```cpp
template<typename T>
class Vector {
   size_t size_;
   size_t cap_;
   T* data_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        cap_ = initial_capacity;
        data_ = new T[cap_];
    }

    T& operator[](size_t index) {
       return data_[index]->size;
    }
};
```

```cpp
Vector<int> v1{16};
int x = v1[10];
```

```
In instantiation of
'T& Vector<T>::operator[](size_t)
[with T = int;
      size_t = long long unsigned int]':
error: base operand of '->' is not a pointer
   86 |         return data_[index]->size;
      |                            ^~~~~
```

We have a usage, so, compiler tried to generate operator[] (and failed)

# Template classes

```
template<typename T>
class Vector {
   size_t size_;
   size_t cap_;
   T* data_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        cap_ = initial_capacity;
        data_ = new T[cap_];
    }

    T& operator[](size_t index) {
       return data_[index]->size;
    }
};
```

```
template class Vector<int>;

Vector<int> v1{16};
```

# Template classes

```cpp
template<typename T>
class Vector {
   size_t size_;
   size_t cap_;
   T* data_;
public:
    Vector(size_t initial_capacity) {
         size_ = 0;
         cap_ = initial_capacity;
         data_ = new T[cap_];
    }

    T& operator[](size_t index) {
       return data_[index]->size;
    }
};
```

```cpp
template class Vector<int>;

Vector<int> v1{16};
```

```
In instantiation of
'T& Vector<T>::operator[](size_t)
[with T = int;
      size_t = long long unsigned int]':
error: base operand of '->' is not a pointer
   86 |         return data_[index]->size;
      |                            ^~~~~
```

# Template classes

This is a direct order to compiler: instantiate it now!

```
template<typename T>
class Vector {
   size_t size_;
   size_t cap_;
   T* data_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        cap_ = initial_capacity;
        data_ = new T[cap_];
    }

    T& operator[](size_t index) {
       return data_[index]->size;
    }
};
```

```
template class Vector<int>;

Vector<int> v1{16};
```

```
In instantiation of
'T& Vector<T>::operator[](size_t)
[with T = int;
     size_t = long long unsigned int]':
error: base operand of '->' is not a pointer
   86 |         return data_[index]->size;
      |                            ^~~~~
```

# Template classes

```
template<typename T>
class Vector {
   size_t size_;
   size_t cap_;
   T* data_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        cap_ = initial_capacity;
        data_ = new T[cap_];
    }

    T& operator[](size_t index) {
       return data_[index]->size;
    }
};
```

This is a direct order to compiler: instantiate it now!

```
template class Vector<int>;

Vector<int> v1{16};
```

```
In instantiation of
'T& Vector<T>::operator[](size_t)
[with T = int;
      size_t = long long unsigned int]':
error: base operand of '->' is not a pointer
   86 |         return data_[index]->size;
      |                            ^~~~~
```

It is rarely used as laziness of template classes instantiation is a default and preferable way.

# Template classes: specialization

```
template <typename T, typename U>
struct Pair {
   T first;
   U second;
};
```

# Template classes: specialization

```
template <typename T, typename U>
struct Pair {
   T first;
   U second;
};


template <>
struct Pair<int, int> {
   int first;
   int second;

   int getSum() {
       return first + second;
   }
};
```

usual specialization, will work when both types are ints

# Template classes: specialization

```
template <typename T, typename U>
struct Pair {
   T first;
   U second;
};
```

```
template <>
struct Pair<int, int> {
   int first;
   int second;

   int getSum() {
       return first + second;
   }
};
```

usual specialization, will work when both types are ints

```
Pair<int, int> p2{3, 5};
std::cout << p2.getSum();
```

# Template classes: specialization

```
template <typename T, typename U>
struct Pair {
   T first;
   U second;
};
```

```
template <typename T>
struct Pair<T, T> {
   T first;
   T second;

   Pair(T val): first(val), second(val) {}
};
```

this is partial specializa

still a template class, but one of template parameters reduced

# Template classes: specialization

```
template <typename T, typename U>
struct Pair {
   T first;
   U second;
};
```

```
template <typename T>
struct Pair<T, T> {
   T first;
   T second;

   Pair(T val): first(val), second(val) {}
};
```

this is partial specializa

still a template class, but one of template parameters reduced

```
Pair<double, double> p2(3.14);
```

# Template classes: specialization

```
template <typename T, typename U>
struct Pair {
   T first;
   U second;
};
```

```
template <typename T>
struct Pair<T, T> {
   T first;
   T second;

   Pair(T val): first(val), second(val) {}
};
```

```
template <>
struct Pair<int, int> {
   int first;
   int second;

   int getSum() {
      return first + second;
   }
};
```

# Template classes: specialization

```
template <typename T, typename U>
struct Pair {
    T first;
    U second;
};
```

```
Pair<int, int> p(3);
```

```
template <typename T>
struct Pair<T, T> {
    T first;
    T second;

    Pair(T val): first(val), second(val) {}
};
```

```
template <>
struct Pair<int, int> {
    int first;
    int second;

    int getSum() {
        return first + second;
    }
};
```

Will it compile?

# Template classes: specialization

```
template <typename T, typename U>
struct Pair {
   T first;
   U second;
};
```

```
Pair<int, int> p(3);
```

```
template <typename T>
struct Pair<T, T> {
   T first;
   T second;

   Pair(T val): first(val), second(val) {}
};
```

```
template <>
struct Pair<int, int> {
   int first;
   int second;

   int getSum() {
      return first + second;
   }
};
```

# Template classes: specialization

Will it compile? Which specialization will be chosen?

```
template <typename T, typename U>
struct Pair {
   T first;
   U second;
};
```

```
Pair<int, int> p(3);
```

```
template <typename T>
struct Pair<T, T> {
   T first;
   T second;

   Pair(T val): first(val), second(val) {}
};
```

```
template <>
struct Pair<int, int> {
   int first;
   int second;

   int getSum() {
      return first + second;
   }
};
```

# Template classes: specialization

Will it compile? Which specialization will be chosen?

```
template <typename T, typename U>
struct Pair {
    T first;
    U second;
};
```

```
Pair<int, int> p(3);
```

```
<source>:27:24: error: no matching
function for call to 'Pair<int,
int>::Pair(int)'
  27 |     Pair<int, int> p2(3);
```

```
template <typename T>
struct Pair<T, T> {
    T first;
    T second;

    Pair(T val): first(val), second(val) {}
};
```

```
template <>
struct Pair<int, int> {
    int first;
    int second;

    int getSum() {
        return first + second;
    }
};
```

# Template classes: specialization

```
template <typename T, typename U>
struct Pair {
   T first;
   U second;
};


template <typename T>
struct Pair<T, T> {
   T first;
   T second;

   Pair(T val): first(val), second(val) {}
};
```

Which specialization will be chosen?

The general rule: the most specific specialization is chosen.

# Template classes: specialization

```cpp
template <typename T, typename U>
struct Pair {
    T first;
    U second;
};


template <typename T>
struct Pair<T, T> {
    T first;
    T second;

    Pair(T val): first(val), second(val) {}
};
```

Which specialization will be chosen?

The general rule: the most specific specialization is chosen.

Specialization forms a partial order over "more-specialized-than" relation, described here.

# Template classes: specialization

```
template <typename T, typename U>
struct Pair {
    T first;
    U second;
};

template <typename T>
struct Pair<int, T> {
    int first;
    T second;

    void print() {
        std::cout << "key = " << first
             << "; data = " << second
             << std::endl;
    }
};
```

# Template classes: specialization

```
template <typename T, typename U>
struct Pair {
   T first;
   U second;
};

template <typename T>
struct Pair<int, T> {
   int first;
   T second;

   void print() {
       std::cout << "key = " << first
            << "; data = " << second
            << std::endl;
   }
};
```

```
template <typename T>
using PairForTreap = Pair<int, T>;
```

# Template classes: specialization

```cpp
template <typename T, typename U>
struct Pair {
   T first;
   U second;
};

template <typename T>
struct Pair<int, T> {
   int first;
   T second;

   void print() {
       std::cout << "key = " << first
            << "; data = " << second
            << std::endl;
   }
};
```

```cpp
template <typename T>
using PairForTreap = Pair<int, T>;


PairForTreap<float> f{13, 5.23};
f.print();
```

# Template classes: specialization

```
template <typename T, typename U>
struct Pair {
    T first;
    U second;
};

template <typename T>
struct Pair<int, T> {
    int first;
    T second;

    void print() {
        std::cout << "key = " << first
             << "; data = " << second
             << std::endl;
    }
};
```

```
template <typename T>
using PairForTreap = Pair<int, T>;


PairForTreap<float> f{13, 5.23};
f.print();
```

Another example of partial specialization (first type argument is int) + using to just name this specialization.

# Template classes: specialization

```
template <typename T, typename U>
struct Pair {
   T first;
   U second;
};

template <typename T, typename U>
struct Pair<T*, U*> {
   T* first;
   U* second;

   bool has_null_pointers() {
       return first == nullptr || second == nullptr;
   }
};
```

# Template classes: specialization

```
template <typename T, typename U>
struct Pair {
    T first;
    U second;
};

template <typename T, typename U>
struct Pair<T*, U*> {
    T* first;
    U* second;

    bool has_null_pointers() {
        return first == nullptr || second == nullptr;
    }
};
```

```
double d = 3.14;
Pair<Matrix*, double*> p4{nullptr, &d};
std::cout << p4.has_null_pointers();
```

# Template classes: specialization

```
template <typename T, typename U>
struct Pair {
    T first;
    U second;
};

template <typename T, typename U>
struct Pair<T*, U*> {
    T* first;
    U* second;

    bool has_null_pointers() {
        return first == nullptr || second == nullptr;
    }
};
```

```
double d = 3.14;
Pair<Matrix*, double*> p4{nullptr, &d};
std::cout << p4.has_null_pointers();
```

Partial specialization for case of pointers.

# Template classes: specialization

```
template <typename T, typename U>
struct Pair {
   T first;
   U second;
};
```

```
template <typename T, typename U>
struct Pair<T*, U*> {
   T* first;
   U* second;

   bool has_null_pointers() {
       return first == nullptr ||
              second == nullptr;
   }
};
```

```
template <typename T>
struct Pair<T, T> {
   T first;
   T second;

   Pair(T val): first(val),
                second(val) {}
};
```

Which specialization will be chosen?

# Template classes: specialization

```
template <typename T, typename U>
struct Pair {
    T first;
    U second;
};
```

```
double d = 3.14;
Pair<double*, double*> p4{nullptr, &d};
```

```
template <typename T, typename U>
struct Pair<T*, U*> {
    T* first;
    U* second;

    bool has_null_pointers() {
        return first == nullptr ||
               second == nullptr;
    }
};
```

```
template <typename T>
struct Pair<T, T> {
    T first;
    T second;

    Pair(T val): first(val),
                 second(val) {}
};
```

Which specialization will be chosen?

# Template classes: specialization

```
template <typename T, typename U>
struct Pair {
    T first;
    U second;
};

template <typename T, typename U>
struct Pair<T*, U*> {
    T* first;
    U* second;

    bool has_null_pointers() {
        return first == nullptr ||
               second == nullptr;
    }
};
```

```
double d = 3.14;
Pair<double*, double*> p4{nullptr, &d};
```

```
error: ambiguous template instantiation for
'struct Pair<double*, double*>'
```

```
template <typename T>
struct Pair<T, T> {
    T first;
    T second;

    Pair(T val): first(val),
                 second(val) {}
};
```

# Template classes: specialization

```cpp
template <typename T, typename U>
struct Pair {
   T first;
   U second;
};


template <typename T>
struct Pair<Vector<T>, Vector<T>> {
   T first_element_left;
   T first_element_right;

   Pair(Vector<T>& v1, Vector<T>& v2) {
       first_element_left = v1[0];
       first_element_right = v2[0];
   }
};
```

# Template classes: specialization

```
template <typename T, typename U>
struct Pair {
    T first;
    U second;
};


template <typename T>
struct Pair<Vector<T>, Vector<T>> {
    T first_element_left;
    T first_element_right;

    Pair(Vector<T>& v1, Vector<T>& v2) {
        first_element_left = v1[0];
        first_element_right = v2[0];
    }
};
```

```
Vector<int> v1;
Vector<int> v2;

Pair<Vector<int>,
     Vector<int>> pv(v1, v2);


Vector<float> v3;
Vector<float> v4;
Pair<Vector<float>,
     Vector<float>> pv2(v3, v4);
```

Specialization will work for any variants of vectors

# Templates: non-type parameters

# Templates: non-type parameters

```
template<typename T>
class Vector {
   size_t size_;
   size_t cap_;
   T* data_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        cap_ = initial_capacity;
        data_ = new T[cap_];
    }

    T& operator[](size_t index) {
       return data_[index];
    }
};
```

# Templates: non-type parameters

```
template<typename T>
class Vector {
   size_t size_;
   size_t cap_;
   T* data_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        cap_ = initial_capacity;
        data_ = new T[cap_];
    }

    T& operator[](size_t index) {
       return data_[index];
    }
};
```

What if we want to work with vectors, which capacity is fixed and known in compile time?

# Templates: non-type parameters

```
template<typename T, int cap>
class Vector {
   size_t size_;
   size_t cap_;
   T* data_;
public:
    Vector(size_t initial_capacity) {
        size_ = 0;
        cap_ = initial_capacity;
        data_ = new T[cap_];
    }

    T& operator[](size_t index) {
       return data_[index];
    }
};
```

What if we want to work with vectors, which capacity is fixed and known in compile time?

# Templates: non-type parameters

```
template<typename T, int cap>
class FixedSizeVector {
   size_t size_;
   T* data_;
public:
    FixedSizeVector () {
        size_ = 0;
        data_ = new T[cap];
    }

    T& operator[](size_t index) {
       return data_[index];
    }
};
```

What if we want to work with vectors, which capacity is fixed and known in compile time?

# Templates: non-type parameters

```
template<typename T, int cap>
class FixedSizeVector {
   size_t size_;
   T* data_;
public:
    FixedSizeVector () {
        size_ = 0;
        data_ = new T[cap];
    }

    T& operator[](size_t index) {
       return data_[index];
    }
};
```

What if we want to work with vectors, which capacity is fixed and known in compile time?

```
FixedSizeVector<int, 16> fsv;
FixedSizeVector<float, 2> fsv2;
```

# Templates: non-type parameters

```
template<typename T, int cap>
class FixedSizeVector {
   size_t size_;
   T* data_;
public:
    FixedSizeVector () {
        size_ = 0;
        data_ = new T[cap];
    }

    T& operator[](size_t index) {
       return data_[index];
    }
};
```

What if we want to work with vectors, which capacity is fixed and known in compile time?

```
FixedSizeVector<int, 16> fsv;
FixedSizeVector<float, 2> fsv2;
```

Works as usual: for each actual value of *cap*, new class will be generated.

# Templates: non-type parameters

```
template<typename T, int cap>
class FixedSizeVector {
   size_t size_;
   T* data_;
public:
    FixedSizeVector () {
        size_ = 0;
        data_ = new T[cap];
    }

    T& operator[](size_t index) {
       return data_[index];
    }
};
```

What if we want to work with vectors, which capacity is fixed and known in compile time?

```
FixedSizeVector<int, 16> fsv;
FixedSizeVector<float, 2> fsv2;
```

Works as usual: for each actual value of *cap*, new class will be generated.

Of course it is dangerous! Type sets are limited, set of values - not really (it is huge).

# Templates: non-type parameters

```
template<typename T, int cap>
class FixedSizeVector {
   size_t size_;
   T* data_;
public:
    FixedSizeVector () {
        size_ = 0;
        data_ = new T[cap];
    }

    T& operator[](size_t index) {
       return data_[index];
    }
};

template<typename T> class FixedSizeVector <T, 0>;
```

# Templates: non-type parameters

```
template<typename T, int cap>
class FixedSizeVector {
   size_t size_;
   T* data_;
public:
    FixedSizeVector () {
        size_ = 0;
        data_ = new T[cap];
    }

    T& operator[](size_t index) {
       return data_[index];
    }
};

template<typename T> class FixedSizeVector <T, 0>;
```

Specialization over non-type parameters still works.

# Templates: non-type parameters

```
template<typename T, int cap>
class FixedSizeVector {
   size_t size_;
   T* data_;
public:
    FixedSizeVector () {
        size_ = 0;
        data_ = new T[cap];
    }

    T& operator[](size_t index) {
       return data_[index];
    }
};

template<typename T> class FixedSizeVector <T, 0>;
```

Specialization over non-type parameters still works. This one is an analogue of "=delete;" for functions specialization.

# Templates: non-type parameters

```
template<typename T, int cap>
class FixedSizeVector {
   size_t size_;
   T* data_;
public:
    FixedSizeVector () {
        size_ = 0;
        data_ = new T[cap];
    }

    T& operator[](size_t index) {
       return data_[index];
    }
};

template<typename T> class FixedSizeVector <T, 0>;
```

```
FixedSizeVector<int, 0> fsv;
```

```
error: aggregate FixedSizeVector<int, 0>
fsv' has incomplete type and cannot be
defined
```

Specialization over non-type parameters still works. This one is an analogue of "=delete;" for functions specialization.

# Templates: compile time execution

# Templates: compile time execution

So, we can define a template with non-type (for example numeric) parameter...

# Templates: compile time execution

So, we can define a template with non-type (for example numeric) parameter… but also specialize it for some values.

# Templates: compile time execution

So, we can define a template with non-type (for example numeric) parameter… but also specialize it for some values.

Sounds like a good combination for recursion!

# Templates: compile time execution (teaser)

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

# Templates: compile time execution (teaser)

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

template <> int fib<1>() { return 1; }
template <> int fib<2>() { return 1; }
```

# Templates: compile time execution (teaser)

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

template <> int fib<1>() { return 1; }
template <> int fib<2>() { return 1; }

int main() {
    std::cout << fib<4>();
    return 0;
}
```

# Templates: compile time execution (teaser)

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

template <> int fib<1>() { return 1; }
template <> int fib<2>() { return 1; }

int main() {
   std::cout << fib<4>();
   return 0;
}
```

If compiler will be very straightforward here, it will generate 4 specializations of fib: fib<1>, fib<2>, fib<3>, fib<4> and just run them

(nothing interesting here)

https://godbolt.org/z/rxYGjve86

# Templates: compile time execution (teaser)

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

template <> int fib<1>() { return 1; }
template <> int fib<2>() { return 1; }

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

```
If compiler will be very
straightforward here, it
will generate 4
specializations of fib:
fib<1>, fib<2>, fib<3>,
fib<4>, …, fib<900> and
just run them

(nothing interesting here,
just exponential complexity,
good like with waiting for
result)
```

https://godbolt.org/z/rxYGjve86

# Templates: compile time execution (teaser)

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

template <> int fib<1>() { return 1; }
template <> int fib<2>() { return 1; }

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

But if this is an optimizing compiler…

# Templates: compile time execution (teaser)

```
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;
}
```

But if this is an optimizing compiler and maybe with some help from us…

inline is hint for compiler: please try to place the code of this function directly in caller

# 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;
}
```

But if this is an optimizing compiler and maybe with some help from us the magic will happen:

```
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
```

https://godbolt.org/z/3YrdvdYe9

# 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;
}
```

But if this is an optimizing compiler and maybe with some help from us the magic will happen:

```
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
```

https://godbolt.org/z/3YrdvdYe9

# Templates: compile time execution (teaser)

```
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;
}
```

1) Compiler inlined each and every template function body inside it’s callers

# Templates: compile time execution (teaser)

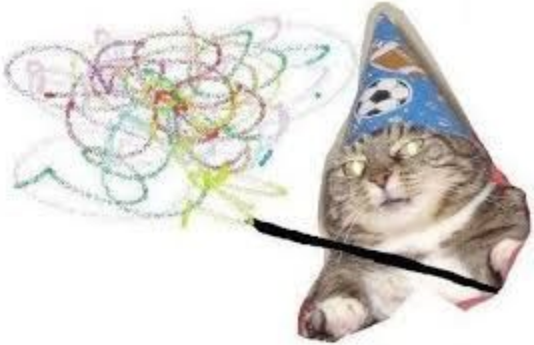
```
inline int fib_1() { return 1; }

inline int fib_2() { return 1; }

inline int fib_3() { return fib_2() + fib_1(); }

inline int fib_4() { return fib_3() + fib_2(); }

void main() {
    cout << fib_4();
}
```

# Templates: compile time execution (teaser)

```
inline int fib_2() { return 1; }

inline int fib_3() { return fib_2() + 1; }

inline int fib_4() { return fib_3() + fib_2(); }

void main() {
    cout << fib_4();
}
```

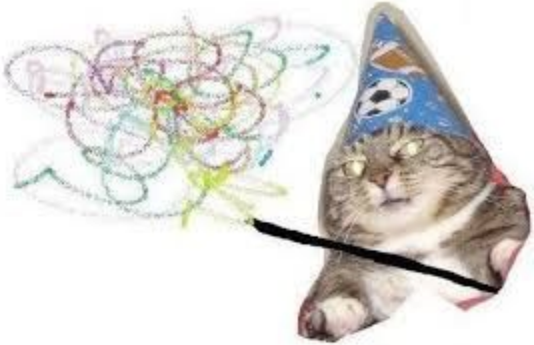
# Templates: compile time execution (teaser)

```
inline int fib_3() { return 1 + 1; }

inline int fib_4() { return fib_3() + 1; }

void main() {
    cout << fib_4();
}
```



# Templates: compile time execution (teaser)

```
inline int fib_4() { return 1 + 1 + 1; }

void main() {
    cout << fib_4();
}
```



# Templates: compile time execution (teaser)

```
void main() {
    cout << 1 + 1 + 1;
}
```

# Templates: compile time execution (teaser)

```
void main() {
    cout << 3;
}
```

# 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;
}
```

1) Compiler inlined each and every template function body inside it's callers

```
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)

```
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;
}
```

1) Compiler inlined each and every template function body inside it's callers

2) Removed bodies of all inlined specializations (except explicit one)

# Templates: compile time execution (teaser)

```
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;
}
```

1) Compiler inlined each and every template function body inside it's callers

2) Removed bodies of all inlined specializations (except explicit one)

3) Didn't itself suffer from exponential complexity! Why?

# Templates: compile time execution (teaser)

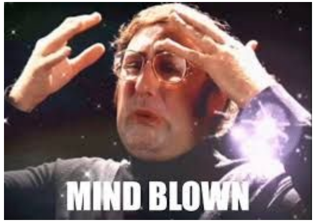
```cpp
template <int N>
inline int fib() {
   return fib<N-1>() + fib<N-2>();
}

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

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

1) Compiler inlined each and every template function body inside it's callers

2) Removed bodies of all inlined specializations (except explicit one)

3) Didn't itself suffer from exponential complexity! Because templates instantiations are cached (hello, memoization!)

# 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)

The same can be done much easier (and without currying favor with the compiler).

# Templates: compile time execution (teaser #2)

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

# Templates: compile time execution (teaser #2)

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

```
template<int X, int Y> // X^Y
struct Pow {
   static const int result = ???;
};
```

# Templates: compile time execution (teaser #2)

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

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

# Templates: compile time execution (teaser #2)

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

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

# Templates: compile time execution (teaser #2)

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

```
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;
};
```

# Templates: compile time execution (teaser #2)

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

```
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;
};
```

# 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;
}
```

# Templates: compile time execution (teaser #2)

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

```
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;
}
```

```
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 and inheritence

# Templates and inheritence

Templates as well as inheritance is an instrument to write a polymorphic code.

# Templates and inheritence

Templates as well as inheritance is an instrument to write a polymorphic code.

Inheritance allows you to use subtyping polymorphism. It is a dynamic polymorphism (implemented via virtual calls).

# Templates and inheritence

Templates as well as inheritance is an instrument to write a polymorphic code.

Inheritance allows you to use subtyping polymorphism. It is a dynamic polymorphism (implemented via virtual calls).

Templates are very different. Because of specializations, each instantiation of a template can be very different from the primary template. So, no LSP.

# Templates and inheritence

Templates as well as inheritance is an instrument to write a polymorphic code.

Inheritance allows you to use subtyping polymorphism. It is a dynamic polymorphism (implemented via virtual calls).

Templates are very different. Because of specializations, each instantiation of a template can be very different from the primary template. So, no LSP.

On the other hand, templates define static polymorphism, that is evaluated completely in compile time.

# Templates and inheritence

Templates as well as inheritance is an instrument to write a polymorphic code.

Inheritance allows you to use subtyping polymorphism. It is a dynamic polymorphism (implemented via virtual calls).

Templates are very different. Because of specializations, each instantiation of a template can be very different from the primary template. So, no LSP.

On the other hand, templates define static polymorphism, that is evaluated completely in compile time.

But what if combine both approaches?

# CRTP via Templates

# CRTP via Templates

Task: we need a mechanism to count instances of a class.

# CRTP via Templates

Task: we need a mechanism to count instances of any class.

```
template <typename T>
struct Counter {









};
```

```cpp
template <typename T>
struct Counter {
   static int objects_created;
   static int objects_alive;

   Counter() {
       ++objects_created;
       ++objects_alive;
   }

   Counter(const Counter&) {
       ++objects_created;
       ++objects_alive;
   }
protected:
   ~Counter() {
       --objects_alive;
   }
};
```

```cpp
template <typename T>
struct Counter {
   static int objects_created;
   static int objects_alive;

   Counter() {
       ++objects_created;
       ++objects_alive;
   }

   Counter(const Counter&) {
       ++objects_created;
       ++objects_alive;
   }
protected:
   ~Counter() {
       --objects_alive;
   }
};

template <typename T> int Counter<T>::objects_created(0);
template <typename T> int Counter<T>::objects_alive(0);
```

```cpp
template <typename T>
struct Counter {
    static int objects_created;
    static int objects_alive;

    Counter() {
        ++objects_created;
        ++objects_alive;
    }

    Counter(const Counter&) {
        ++objects_created;
        ++objects_alive;
    }
protected:
    ~Counter() {
        --objects_alive;
    }
};

template <typename T> int Counter<T>::objects_created(0);
template <typename T> int Counter<T>::objects_alive(0);
```

Strange, we have T, but it is not used!

```cpp
template <typename T>
struct Counter {
    static int objects_created;
    static int objects_alive;

    Counter() {
        ++objects_created;
        ++objects_alive;
    }

    Counter(const Counter&) {
        ++objects_created;
        ++objects_alive;
    }
protected:
    ~Counter() {
        --objects_alive;
    }
};
```

```cpp
class Foo: Counter<Foo> {};
class Bar: Counter<Bar> {};
```

```cpp
template <typename T>
struct Counter {
    static int objects_created;
    static int objects_alive;

    Counter() {
        ++objects_created;
        ++objects_alive;
    }

    Counter(const Counter&) {
        ++objects_created;
        ++objects_alive;
    }
protected:
    ~Counter() {
        --objects_alive;
    }
};
```

```cpp
class Foo: Counter<Foo> {};
class Bar: Counter<Bar> {};
```

Instantiation of Counter for Foo and Bar

```cpp
template <typename T>
struct Counter {
   static int objects_created;
   static int objects_alive;

   Counter() {
       ++objects_created;
       ++objects_alive;
   }

   Counter(const Counter&) {
       ++objects_created;
       ++objects_alive;
   }
protected:
   ~Counter() {
       --objects_alive;
   }
};
```

```cpp
class Foo: Counter<Foo> {};
class Bar: Counter<Bar> {};
```

Instantiation of Counter for Foo and Bar

Static vars are created for each instantiation

```cpp
template <typename T>
struct Counter {
   static int objects_created;
   static int objects_alive;

   Counter() {
        ++objects_created;
        ++objects_alive;
   }

   Counter(const Counter&) {
        ++objects_created;
        ++objects_alive;
   }
protected:
   ~Counter() {
        --objects_alive;
   }
};
```

```cpp
class Foo: Counter<Foo> {};
class Bar: Counter<Bar> {};
```

Instantiation of Counter for Foo and Bar

Static vars are created for each instantiation

```cpp
int main() {
    {
        Foo f;
        Foo *pf = new Foo();
        Bar b;
    }
    return 0;
}
```

```cpp
template <typename T>
struct Counter {
    static int objects_created;
    static int objects_alive;

    Counter() {
        ++objects_created;
        ++objects_alive;
    }

    Counter(const Counter&) {
        ++objects_created;
        ++objects_alive;
    }
protected:
    ~Counter() {
        --objects_alive;
    }
};
```

```cpp
class Foo: Counter<Foo> {};
class Bar: Counter<Bar> {};
```

Instantiation of Counter for Foo and Bar

Static vars are created for each instantiation

```cpp
int main() {
    {
        Foo f;
        Foo *pf = new Foo();
        Bar b;
        cout << Counter<Foo>::objects_created;
        cout << Counter<Foo>::objects_alive;
        cout << Counter<Bar>::objects_created;
        cout << Counter<Bar>::objects_alive;
    }
    return 0;
}
```

```cpp
template <typename T>
struct Counter {
    static int objects_created;
    static int objects_alive;

    Counter() {
        ++objects_created;
        ++objects_alive;
    }

    Counter(const Counter&) {
        ++objects_created;
        ++objects_alive;
    }
protected:
    ~Counter() {
        --objects_alive;
    }
};
```

```cpp
class Foo: Counter<Foo> {};
class Bar: Counter<Bar> {};
```

Instantiation of Counter for Foo and Bar

Static vars are created for each instantiation

```cpp
int main() {
    {
        Foo f;
        Foo *pf = new Foo();
        Bar b;
        cout << Counter<Foo>::objects_created; // 2
        cout << Counter<Foo>::objects_alive;   // 2
        cout << Counter<Bar>::objects_created; // 1
        cout << Counter<Bar>::objects_alive;   // 1
    }
    return 0;
}
```

```cpp
template <typename T>
struct Counter {
    static int objects_created;
    static int objects_alive;

    Counter() {
        ++objects_created;
        ++objects_alive;
    }

    Counter(const Counter&) {
        ++objects_created;
        ++objects_alive;
    }
protected:
    ~Counter() {
        --objects_alive;
    }
};
```

```cpp
class Foo: Counter<Foo> {};
class Bar: Counter<Bar> {};
```

Instantiation of Counter for Foo and Bar

Static vars are created for each instantiation

```cpp
int main() {
    {
        Foo f;
        Foo *pf = new Foo();
        Bar b;
        cout << Counter<Foo>::objects_created; // 2
        cout << Counter<Foo>::objects_alive;   // 2
        cout << Counter<Bar>::objects_created; // 1
        cout << Counter<Bar>::objects_alive;   // 1
    }
    cout << Counter<Foo>::objects_created;
    cout << Counter<Foo>::objects_alive;
    cout << Counter<Bar>::objects_created;
    cout << Counter<Bar>::objects_alive;
    return 0;
}
```

```cpp
template <typename T>
struct Counter {
   static int objects_created;
   static int objects_alive;

   Counter() {
       ++objects_created;
       ++objects_alive;
   }

   Counter(const Counter&) {
       ++objects_created;
       ++objects_alive;
   }
protected:
   ~Counter() {
       --objects_alive;
   }
};
```

```cpp
class Foo: Counter<Foo> {};
class Bar: Counter<Bar> {};
```

Instantiation of Counter for Foo and Bar

Static vars are created for each instantiation

```cpp
int main() {
    {
        Foo f;
        Foo *pf = new Foo();
        Bar b;
        cout << Counter<Foo>::objects_created; // 2
        cout << Counter<Foo>::objects_alive;   // 2
        cout << Counter<Bar>::objects_created; // 1
        cout << Counter<Bar>::objects_alive;   // 1
    }
    cout << Counter<Foo>::objects_created;     // 2
    cout << Counter<Foo>::objects_alive;       // 1
    cout << Counter<Bar>::objects_created;     // 1
    cout << Counter<Bar>::objects_alive;       // 0
    return 0;
}
```

```cpp
template <typename T>
struct Counter {
   static int objects_created;
   static int objects_alive;

   Counter() {
       ++objects_created;
       ++objects_alive;
   }

   Counter(const Counter&) {
       ++objects_created;
       ++objects_alive;
   }
protected:
   ~Counter() {
       --objects_alive;
   }
};
```

```cpp
class Foo: Counter<Foo> {};
class Bar: Counter<Bar> {};


class Vector: Counter<Vector> {
     ...
};
```

Instantiation of Counter for Foo and Bar

```cpp
template <typename T>
struct Counter {
    static int objects_created;
    static int objects_alive;

    Counter() {
        ++objects_created;
        ++objects_alive;
    }

    Counter(const Counter&) {
        ++objects_created;
        ++objects_alive;
    }
protected:
    ~Counter() {
        --objects_alive;
    }
};
```

```cpp
class Foo: Counter<Foo> {};
class Bar: Counter<Bar> {};
```

Instantiation of Counter for Foo and Bar

```cpp
template <typename T>
class Vector: Counter<Vector<T>> {
    ...
};
```

```cpp
template <typename T>
struct Counter {
   static int objects_created;
   static int objects_alive;

   Counter() {
       ++objects_created;
       ++objects_alive;
   }

   Counter(const Counter&) {
       ++objects_created;
       ++objects_alive;
   }
protected:
   ~Counter() {
       --objects_alive;
   }
};
```

```cpp
class Foo: Counter<Foo> {};
class Bar: Counter<Bar> {};
```

Instantiation of Counter for Foo and Bar

```cpp
template <typename T>
class Vector: Counter<Vector<T>> {
    ...
};
```

Counter will be instantiated for each instantiation of Vector!

```cpp
template <typename T>
struct Counter {
   static int objects_created;
   static int objects_alive;

   Counter() {
       ++objects_created;
       ++objects_alive;
   }

   Counter(const Counter&) {
       ++objects_created;
       ++objects_alive;
   }
protected:
   ~Counter() {
       --objects_alive;
   }
};
```

```cpp
class Foo: Counter<Foo> {};
class Bar: Counter<Bar> {};
```

Instantiation of Counter for Foo and Bar

```cpp
template <typename T>
class Vector: Counter<Vector<T>> {
    ...
};
```

Counter will be instantiated for each instantiation of Vector!

Such trick is called CRTP:
Curiously Recurring Template Pattern

```cpp
template <typename T>
struct Counter {
    static int objects_created;
    static int objects_alive;

    Counter() {
        ++objects_created;
        ++objects_alive;
    }

    Counter(const Counter&) {
        ++objects_created;
        ++objects_alive;
    }
protected:
    ~Counter() {
        --objects_alive;
    }
};
```

```cpp
class Foo: Counter<Foo> {};
class Bar: Counter<Bar> {};
```

Instantiation of Counter for Foo and Bar

```cpp
template <typename T>
class Vector: Counter<Vector<T>> {
    ...
};
```

Counter will be instantiated for each instantiation of Vector!

Such trick is called CRTP:
Curiously Recurring Template Pattern

And it is much more powerful!

# CRTP via Templates

CRTP allows you to mix-in new functionality into your classes.

# CRTP via Templates

CRTP allows you to mix-in new functionality into your classes.

Actually, it is not about "is-a" relationship, so, we use private inheritance here.

# CRTP via Templates

CRTP allows you to mix-in new functionality into your classes.

Actually, it is not about "is-a" relationship, so, we use private inheritance here.

Works really well thanks to multiple inheritance: you can mix-in a lot of features into your classes because of it.

# CRTP via Templates

CRTP allows you to mix-in new functionality into your classes.

Actually, it is not about "is-a" relationship, so, we use private inheritance here.

Works really well thanks to multiple inheritance: you can mix-in a lot of features into your classes because of it.

But that's not all! You can simulate subtyping polymorphism and virtual calls with CRTP.

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

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

Here we specify that:

1. Derived must have print_impl method

2. Person instance can be casted into Derived statically.

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

class Student: public Person<Student> {
public:
   void print_impl() {
       std::cout << "student print";
   }
};
```

Here we specify that:

1. Derived must have print_impl method
2. Person instance can be casted into Derived statically.

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

class Student: public Person<Student> {
public:
   void print_impl() {
       std::cout << "student print";
   }
};

template<typename Derived>
void polymorphic_print(Person<Derived>& p) {
   p.print();
}
```

Here we specify that:

1. Derived must have print_impl method
2. Person instance can be casted into Derived statically.

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

class Student: public Person<Student> {
public:
   void print_impl() {
       std::cout << "student print";
   }
};

template<typename Derived>
void polymorphic_print(Person<Derived>& p) {
   p.print();
}
```

Here we specify that:

1. Derived must have print_impl method

2. Person instance can be casted into Derived statically.

Student& can be substituted here

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

class Student: public Person<Student> {
public:
   void print_impl() {
       std::cout << "student print";
   }
};

template<typename Derived>
void polymorphic_print(Person<Derived>& p) {
   p.print();
}

Student s;
polymorphic_print(s);
```

Here we specify that:

1. Derived **must** have print_impl method

2. Person instance can be casted into Derived **statically**.

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

class Student: public Person<Student> {
public:
   void print_impl() {
       std::cout << "student print";
   }
};

template<typename Derived>
void polymorphic_print(Person<Derived>& p) {
   p.print();
}

Student s;
polymorphic_print(s);
```

← in some situations type here can be omitted

Here we specify that:

1. Derived must have print_impl method
2. Person instance can be casted into Derived statically.

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

class Student: public Person<Student> {
public:
   void print_impl() {
       std::cout << "student print";
   }
};

template<typename Derived>
void polymorphic_print(Person<Derived>& p) {
   p.print();
}

Student s;
polymorphic_print(s);
```

in some situations type here can be omitted

Here we specify that:

1. Derived must have print_impl method
2. Person instance can be casted into Derived statically.

Here we have the same behaviour as with virtual calls, but for free (however, not so convenient, as many things should be done manually)

# Further directions

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

# Further directions

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

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

# 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

# Not So Tiny Task №9 (1 point)

Generalize data structure that you've implemented in NSTT #1, #3 and #4 with help of templates!

## Not So Tiny Task №9 (1 point)

Generalize data structure that you've implemented in NSTT #1, #3 and #4 with help of templates!

## Not So Tiny Task №10 (1 point)

Implement a mix-in to limit number of your class instances. Classes with such functionality should be successfully created only when there are less than specified number of instances, otherwise their construction should fail.
Use CRTP and non-type template argument (for the limit)