Not So Tiny Task №14 (1 point)

Implement a function:

```
template<size_t SIZE, typename... Types>
void allocate(void* memory, Types... args) { ... }
```

That takes some preallocated memory of the given *SIZE*, and variadic number of arguments of different types. Than it initializes new elements of the corresponding types inside the given memory as copies of the given values. Use placement new for that.

Not So Tiny Task №14 (1 point)

Implement a function:

```
template<size_t SIZE, typename... Types>
void allocate(void* memory, Types... args) { ... }
```

That takes some preallocated memory of the given SIZE, and variadic number of arguments of different types. Than it initializes new elements of the corresponding types inside the given memory as copies of the given values. Use placement new for that.

It should be statically checked that SIZE is enough for such allocation.

It should be statically checked that all types are copy_constructable.

Not So Tiny Task №15 (1 point)

Implement a container:

```
template<typename... Types>
class Container {
    ...

public:
    Container(Types... args) { … }

    template<typename T>
    T getElement(size_t idx) { … }
};
```

That encapsulates memory storage for all given arguments (placed in some memory sequentially, one by one).

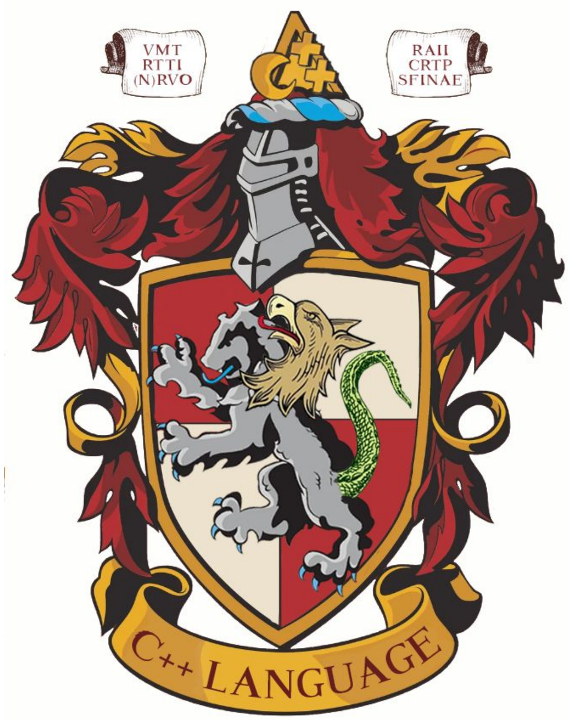
Not So Tiny Task №15 (1 point)

Example:

```
Container<int, char, Point> c(12, 'c', Point{2 ,3});
std::cout << c.getElement<int>(0) << std::endl;
std::cout << c.getElement<char>(1) << std::endl;
std::cout << c.getElement<Point>(2) << std::endl;
```

System Programming with C++

constexpr, concepts



Compile time requirements

Where compile time constants are needed in C++?

Compile time requirements

Where compile time constants are needed in C++?

```
int main() {
    int array[3];
    for (size_t idx = 0; idx < 3; idx++) {
        array[idx] = idx * idx;
    }
    return 0;
}
```

Compile time requirements

Where compile time constants are needed in C++?

size of static array must be compile time constant

```
int main() {
    int array[3];
    for (size_t idx = 0; idx < 3; idx++) {
        array[idx] = idx * idx;
    }
    return 0;
}
```

Compile time requirements

Where compile time constants are needed in C++?

size of static array must be compile time constant
(we are not talking about VLA here of course)

```
int main() {
    int array[3];
    for (size_t idx = 0; idx < 3; idx++) {
        array[idx] = idx * idx;
    }
    return 0;
}
```

Compile time requirements

Where compile time constants are needed in C++?

```
template <typename T, int SIZE>
class LimitedArray {
    T* values;
public:
    LimitedArray(): values(new T[SIZE]) {}
    ~LimitedArray() { delete[] values; }
    T& operator[] (const size_t idx) {
        if (idx > SIZE) {
            throw std::out_of_range("...");
        }
        return values[idx];
    }
};
```

Compile time requirements

Where compile time constants are needed in C++?

```cpp
template <typename T, int SIZE>
class LimitedArray {
    T* values;
public:
    LimitedArray(): values(new T[SIZE]) {}
    ~LimitedArray() { delete[] values; }
    T& operator[] (const size_t idx) {
        if (idx > SIZE) {
            throw std::out_of_range("...");
        }
        return values[idx];
    }
};
```

```cpp
int main() {
    LimitedArray<int, 16> larr{};
    ...
    return 0;
}
```

Compile time requirements

non-typename template arguments also must be constant (how else compile-time specialization will work?)

Where compile time constants are needed in C++?

```
template <typename T, int SIZE>
class LimitedArray {
    T* values;
public:
    LimitedArray(): values(new T[SIZE]) {}
    ~LimitedArray() { delete[] values; }
    T& operator[] (const size_t idx) {
        if (idx > SIZE) {
            throw std::out_of_range("...");
        }
        return values[idx];
    }
};
```

```
int main() {
    LimitedArray<int, 16> larr{};
    ...
    return 0;
}
```

Compile time requirements

Where compile time constants are needed in C++?

```cpp
#include <bitset>

int main() {

    std::bitset<64> mask;
    ...
    return 0;

}
```

same situation: integral template argument
(must be compile time constant)

Compile time requirements

But what is compile time constant in C++?

```
#include <bitset>

int main() {

    std::bitset<64> mask;
    ...
    return 0;

}
```

Compile time requirements

But what is compile time constant in C++?

First of all: literals.

```
#include <bitset>

int main() {

    std::bitset<64> mask;
    ...
    return 0;

}
```

Compile time requirements

But what is compile time constant in C++?

```
#include <bitset>

int main() {

    std::bitset<64> mask;
    ...
    return 0;

}
```

First of all: literals.

```
42;
0xff;
0xb101001;
3.14f;
'P';
...
```

Compile time requirements

But what is compile time constant in C++?

```
#include <bitset>

int main() {

    std::bitset<64> mask;
    ...
    return 0;

}
```

First of all: literals.

```
int   x = 42;
int   y = 0xff;
int   b = 0xb101001;
float f = 3.14f;
char  c = 'P';
          ...
```

Types for which we have literals are called LiteralType

Compile time requirements

But what is compile time constant in C++?

```
#include <bitset>

int main() {

    std::bitset<0x40> mask;
    ...
    return 0;

}
```

First of all: literals.

```
int   x = 42;
int   y = 0xff;
int   b = 0xb101001;
float f = 3.14f;
char  c = 'P';
          ...
```

Types for which we have literals are called LiteralType

Compile time requirements

But what is compile time constant in C++?

```
#include <bitset>

int main() {

    std::bitset<8 * 8> mask;
    ...
    return 0;

}
```

First of all: literals.

Also, arithmetic operations on them.

Compile time requirements

But what is compile time constant in C++?

```
#include <bitset>

int main() {

    std::bitset<sizeof(int)> mask;
    ...
    return 0;

}
```

First of all: literals.

Also, arithmetic operations on them.

Also, call of special functions, like sizeof, ...

Compile time requirements

But what is compile time constant in C++?

```
#include <bitset>

int main() {

    std::bitset<sizeof(int)> mask;
    ...
    return 0;

}
```

First of all: literals.

Also, arithmetic operations on them.

Also, call of special functions, like sizeof, …

Also, elements of enums.

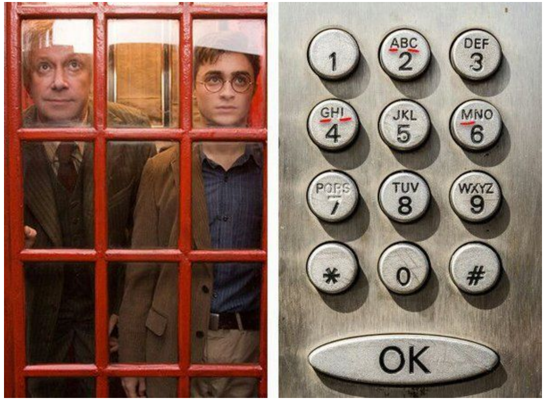
Compile time requirements

Ok, but we need to somehow fight with magic numbers

```
int main() {

    std::bitset<64> mask;
    for (size_t idx = 0; idx < 64; idx++) {
        mask[idx] = (idx % 2);
    }
    std::cout << mask;

    return 0;
}
```



Compile time requirements

Ok, but we need to somehow fight with magic numbers and copy-paste if we have some non-trivial logic in the expression!

```
int main() {

    std::bitset<sizeof(int) << 2> mask;
    for (size_t idx = 0; idx <sizeof(int) << 2; idx++) {
        mask[idx] = (idx % 2);
    }
    std::cout << mask;

    return 0;
}
```

Compile time requirements

Ok, but we need to somehow fight with magic numbers and copy-paste if we have some non-trivial logic in the expression! We can always use macroses, but this is not quite C++ way.

```
int main() {

    std::bitset<sizeof(int) << 2> mask;
    for (size_t idx = 0; idx <sizeof(int) << 2; idx++) {
        mask[idx] = (idx % 2);
    }
    std::cout << mask;

    return 0;
}
```

Compile time requirements

Ok, but we need to somehow fight with magic numbers and copy-paste if we have some non-trivial logic in the expression!

```
int main() {

    int size = sizeof(int) << 2;
    std::bitset<size> mask;
    for (size_t idx = 0; idx < size; idx++) {
        mask[idx] = (idx % 2);
    }
    std::cout << mask;

    return 0;
}
```

Compile time requirements

Ok, but we need to somehow fight with magic numbers and copy-paste if we have some non-trivial logic in the expression!

```
int main() {

    int size = sizeof(int) << 2;
    std::bitset<size> mask;
    for (size_t idx = 0; idx < size; idx++) {
        mask[idx] = (idx % 2);
    }
    std::cout << mask;

    return 0;
}
```

```
error: the value of 'size'
is not usable in a constant
expression
```

Compile time requirements

Ok, but we need to somehow fight with magic numbers and copy-paste if we have some non-trivial logic in the expression!

```
int main() {

    const int size = sizeof(int) << 2;
    std::bitset<size> mask;
    for (size_t idx = 0; idx < size; idx++) {
        mask[idx] = (idx % 2);
    }
    std::cout << mask;

    return 0;
}
```

Compile time requirements

Ok, but we need to somehow fight with magic numbers and copy-paste if we have some non-trivial logic in the expression!

```
int main() {

    const int size = sizeof(int) << 2;
    std::bitset<size> mask;
    for (size_t idx = 0; idx < size; idx++) {
        mask[idx] = (idx % 2);
    }
    std::cout << mask;

    return 0;
}
```

This will work: const integral variable which is initialized with constants is constant itself.

Compile time requirements

Ok, but we need to somehow fight with magic numbers and copy-paste if we have some non-trivial logic in the expression!

```cpp
int main() {

    const int size = sizeof(int) << 2;
    std::bitset<size> mask;
    for (size_t idx = 0; idx < size; idx++) {
        mask[idx] = (idx % 2);
    }
    std::cout << mask;

    return 0;
}
```

This will work: const integral variable which is initialized with constants is constant itself.

```asm
main:
        push    rbp
        mov     rbp, rsp
        push    rbx
        sub     rsp, 56
        mov     DWORD PTR [rbp-28], 16
        mov     QWORD PTR [rbp-56], 0
        mov     QWORD PTR [rbp-24], 0
        jmp     .L22
```

Compile time requirements

But does `const` always mean "compile time constant"?

```
int main() {

    const int size = sizeof(int) << 2;
    std::bitset<size> mask;
    for (size_t idx = 0; idx < size; idx++) {
        mask[idx] = (idx % 2);
    }
    std::cout << mask;

    return 0;
}
```

Compile time requirements

But does const always mean "compile time constant"?
Of course not!

```
int main() {
    int n; std::cin >> n;
    const int size = n;

    std::bitset<size> mask;
    for (size_t idx = 0; idx < size; idx++) {
        mask[idx] = (idx % 2);
    }
    std::cout << mask;

    return 0;
}
```

Compile time requirements

But does const always mean "compile time constant"?
Of course not!

```
int main() {
    int n; std::cin >> n;
    const int size = n;

    std::bitset<size> mask;
    for (size_t idx = 0; idx < size; idx++) {
        mask[idx] = (idx % 2);
    }
    std::cout << mask;

    return 0;
}
```

error: the value of 'size' is not usable in a constant expression

Compile time requirements

But does const always mean "compile time constant"?
Of course not!

Const means that you just can't change the value through name size.

```
int main() {
    int n; std::cin >> n;
    const int size = n;

    std::bitset<size> mask;
    for (size_t idx = 0; idx < size; idx++) {
        mask[idx] = (idx % 2);
    }
    std::cout << mask;

    return 0;
}
```

```
error: the value of 'size'
is not usable in a constant
expression
```

Compile time requirements

But does const always mean "compile time constant"?
Of course not!

```
int main() {
    int n; std::cin >> n;
    const int size = n;

    std::bitset<size> mask;
    for (size_t idx = 0; idx < size; idx++) {
        mask[idx] = (idx % 2);
    }
    std::cout << mask;

    return 0;
}
```

Const means that you just can't change the value through name size. It doesn't mean that the value is known during compilation.

```
error: the value of 'size'
is not usable in a constant
expression
```

Compile time requirements

But does const always mean "compile time constant"?
Of course not!

```
unsigned int const volatile * status_reg;
```

Compile time requirements

But does const always mean "compile time constant"?
Of course not!

```
unsigned int const volatile * status_reg;
```

Here we have a pointer to const (you can't change it) and volatile (someone else from outside can change it!) unsigned int.

Compile time requirements

But does const always mean "compile time constant"?
Of course not!

```
unsigned int const volatile * status_reg;
```

Here we have a pointer to const (you can’t change it) and volatile (someone else from outside can change it!) unsigned int.

volatile just means: hey, compiler, please do not optimize it as someone else can change it.

Compile time requirements

But does `const` always mean "compile time constant"?
Of course not!

```
int main() {
    int n; std::cin >> n;
    const int size = n;

    std::bitset<size> mask;
    for (size_t idx = 0; idx < size; idx++) {
        mask[idx] = (idx % 2);
    }
    std::cout << mask;

    return 0;
}
```

Const means that you just can't change the value through name size. It doesn't mean that the value is known during compilation.

```
error: the value of 'size'
is not usable in a constant
expression
```

Compile time requirements

But does `const` always mean "compile time constant"?
Of course not!

```
int get_size() { return sizeof(int) << 2; }

int main() {
    const int size = get_size();
    std::bitset<size> mask;
    for (size_t idx = 0; idx < size; idx++) {
        mask[idx] = (idx % 2);
    }
    std::cout << mask;

    return 0;
}
```

```
error: the value of 'size'
is not usable in a constant
expression
```

Compile time requirements

So, using const with suitable initializer is fragile, it would be cool to have a special way to define compile time constants.

```
int get_size() { return sizeof(int) << 2; }

int main() {
    const int size = get_size();
    std::bitset<size> mask;
    for (size_t idx = 0; idx < size; idx++) {
        mask[idx] = (idx % 2);
    }
    std::cout << mask;

    return 0;
}
```

error: the value of 'size' is not usable in a constant expression

constexpr

```cpp
int main() {

    const int size = sizeof(int) << 2;
    std::bitset<size> mask;
    for (size_t idx = 0; idx < size; idx++) {
        mask[idx] = (idx % 2);
    }
    std::cout << mask;

    return 0;
}
```

constexpr

```cpp
int main() {

    constexpr int size = sizeof(int) << 2;
    std::bitset<size> mask;
    for (size_t idx = 0; idx < size; idx++) {
        mask[idx] = (idx % 2);
    }
    std::cout << mask;

    return 0;
}
```

constexpr

```cpp
int main() {

    constexpr int size = sizeof(int) << 2;
    std::bitset<size> mask;
    for (size_t idx = 0; idx < size; idx++) {
        mask[idx] = (idx % 2);
    }
    std::cout << mask;

    return 0;
}
```

Compiler checks that the initializer of constexpr is compile time constant. And evaluates it in comp time.

constexpr

```
int main() {

    constexpr int size = sizeof(int) << 2;
    std::bitset<size> mask;
    for (size_t idx = 0; idx < size; idx++) {
        mask[idx] = (idx % 2);
    }
    std::cout << mask;

    return 0;
}
```

Compiler checks that the initializer of constexpr is compile time constant. And evaluates it in comp time.

Can be initialized only with LiteralType value.

constexpr

```
struct Point {
    int x, y;
    Point(int x, int y): x(x), y(y) {}
};

int main() {
    constexpr Point p{1, 2};
    ...
    return 0;
}
```

error: constexpr variable cannot have non-literal type 'const Point'

Compiler checks that the initializer of constexpr is compile time constant. And evaluates it in comp time.

Can be initialized only with LiteralType value.

constexpr

```
struct Point {
    int x, y;
};


int main() {
    constexpr Point p{1, 2};
    ...
    return 0;
}
```

This is ok, aggregate types which consist of literal types are literal types as well.

Compiler checks that the initializer of constexpr is compile time constant. And evaluates it in comp time.

Can be initialized only with LiteralType value.

constexpr

Compiler checks that the initializer of constexpr is compile time constant. And evaluates it in comp time.

```
int main() {

    int n; std::cin >> n;
    constexpr int size = n;

    std::bitset<size> mask;
    for (size_t idx = 0; idx < size; idx++) {
        mask[idx] = (idx % 2);
    }
    std::cout << mask;

    return 0;
}
```

constexpr

```
int main() {

    int n; std::cin >> n;
    constexpr int size = n;

    std::bitset<size> mask;
    for (size_t idx = 0; idx < size; idx++) {
        mask[idx] = (idx % 2);
    }
    std::cout << mask;

    return 0;
}
```

Compiler checks that the initializer of constexpr is compile time constant. And evaluates it in comp time.

```
error: the value of 'n' is not usable in a constant expression
```

constexpr

```
int main() {

    int n; std::cin >> n;
    constexpr int size = n;

    std::bitset<size> mask;
    for (size_t idx = 0; idx < size; idx++) {
        mask[idx] = (idx % 2);
    }
    std::cout << mask;

    return 0;
}
```

Compiler checks that the initializer of constexpr is compile time constant. And evaluates it in comp time.

error: the value of 'n' is not usable in a constant expression

constexpr implies const

constexpr

```
int main() {

    int n; std::cin >> n;
    constexpr int size = n;

    std::bitset<size> mask;
    for (size_t idx = 0; idx < size; idx++) {
        mask[idx] = (idx % 2);
    }
    std::cout << mask;

    return 0;
}
```

Compiler checks that the initializer of constexpr is compile time constant. And evaluates it in comp time.

```
error: the value of 'n' is not usable in a constant expression
```

constexpr implies const

but not vice versa!

constexpr

```
int get_size() { return sizeof(int) << 2; }

int main() {
    constexpr int size = get_size();
    std::bitset<size> mask;
    for (size_t idx = 0; idx < size; idx++) {
        mask[idx] = (idx % 2);
    }
    std::cout << mask;

    return 0;
}
```

constexpr

```
int get_size() { return sizeof(int) << 2; }

int main() {
    constexpr int size = get_size();
    std::bitset<size> mask;
    for (size_t idx = 0; idx < size; idx++) {
        mask[idx] = (idx % 2);
    }
    std::cout << mask;

    return 0;
}
```

```
error: call to non-'constexpr'
function 'int get_size()'
```

constexpr

```cpp
constexpr int get_size() { return sizeof(int) << 2; }

int main() {
    constexpr int size = get_size();
    std::bitset<size> mask;
    for (size_t idx = 0; idx < size; idx++) {
        mask[idx] = (idx % 2);
    }
    std::cout << mask;

    return 0;
}
```

In case of function it means:

constexpr

```
constexpr int get_size() {
    return sizeof(int) << 2;
}

int main() {
    constexpr int size = get_size();
    std::bitset<size> mask;
    for (size_t idx = 0; idx < size; idx++) {
        mask[idx] = (idx % 2);
    }
    std::cout << mask;

    return 0;
}
```

constexpr

In case of function it means:

1) Check that function can be evaluated in compile time if all arguments are constants

```
constexpr int get_size(int shift) {
    return sizeof(int) << shift;
}

int main() {
    constexpr int size = get_size(3);
    std::bitset<size> mask;
    for (size_t idx = 0; idx < size; idx++) {
        mask[idx] = (idx % 2);
    }
    std::cout << mask;

    return 0;
}
```

constexpr

```
constexpr int get_size(int shift) {
    return sizeof(int) << shift;
}

int main() {
    constexpr int size = get_size(3);
    std::bitset<size> mask;
    for (size_t idx = 0; idx < size; idx++) {
        mask[idx] = (idx % 2);
    }
    std::cout << mask;

    return 0;
}
```

In case of function it means:

1) Check that function can be evaluated in compile time if all arguments are constants

2) If check is ok, call of such function can be used (with constant arguments) where you need compile time constant

constexpr

```
constexpr int get_size(int shift) {
    return sizeof(int) << shift;
}

int main() {
    constexpr int size = get_size(3);
    std::bitset<size> mask;
    for (size_t idx = 0; idx < size; idx++) {
        mask[idx] = (idx % 2);
    }
    std::cout << mask;

    return 0;
}
```

In case of function it means:

1) Check that function can be evaluated in compile time if all arguments are constants

2) If check is ok, call of such function can be used (with constant arguments) where you need compile time constant

3) And where needed, it will be evaluated in comp time

constexpr

```cpp
#include <iostream>
#include <bitset>

constexpr int get_size(int shift) {
    return sizeof(int) << shift;
}

int main() {
    constexpr int size = get_size(3);
    std::bitset<size> mask;
    for (size_t idx = 0; idx < size; idx++) {
        mask[idx] = (idx % 2);
    }
    std::cout << mask;

    return 0;
}
```

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

```asm
main:
        push    rbp
        mov     rbp, rsp
        push    rbx
        sub     rsp, 56
        mov     DWORD PTR [rbp-28], 32
        mov     QWORD PTR [rbp-56], 0
        mov     QWORD PTR [rbp-24], 0
        jmp     .L20
.L21:
        mov     rax, QWORD PTR [rbp-24]
        and     eax, 1
        test    rax, rax
        setne   al
        movzx   ebx, al
        lea     rax, [rbp-48]
```

https://godbolt.org/z/o57hdjbf9

constexpr

```
constexpr int get_size(int shift) {
    return sizeof(int) << shift;
}

int main() {
    constexpr int size = get_size(3);
    std::bitset<size> mask;
    for (size_t idx = 0; idx < size; idx++) {
        mask[idx] = (idx % 2);
    }
    std::cout << mask;

    return 0;
}
```

In case of function it means:

1) Check that function can be evaluated in compile time if all arguments are constants

2) If check is ok, call of such function can be used (with constant arguments) where you need compile time constant

3) And where needed, it will be evaluated in comp time

constexpr

```cpp
constexpr int get_size(int shift) {
    return sizeof(int) << shift;
}

int main() {
    int n;
    std::cin >> n;
    int x = get_size(n);
    std::cout << x;

    return 0;
}
```

In case of function it means:

1) Check that function can be evaluated in compile time if all arguments are constants

2) If check is ok, call of such function can be used (with constant arguments) where you need compile time constant

3) And where needed, it will be evaluated in comp time

constexpr

```cpp
constexpr int get_size(int shift) {
    return sizeof(int) << shift;
}

int main() {
    int n;
    std::cin >> n;
    int x = get_size(n);
    std::cout << x;

    return 0;
}
```

In case of function it means:

1) Check that function can be evaluated in compile time if all arguments are constants

2) If check is ok, call of such function can be used (with constant arguments) where you need compile time constant

3) And where needed, it will be evaluated in comp time

```asm
mov     eax, DWORD PTR [rbp-8]
mov     edi, eax
call    get_size(int)
mov     DWORD PTR [rbp-4], eax
```

constexpr

```
constexpr int get_size(int shift) {
    return sizeof(int) << shift;
}

int main() {
    int n;
    std::cin >> n;
    int x = get_size(n);
    std::cout << x;

    return 0;
}
```

In case of function it means:

1) Check that function can be evaluated in compile time if all arguments are constants

2) If check is ok, call of such function can be used (with constant arguments) where you need compile time constant

3) And where needed, it will be evaluated in comp time

Still can be used as a usual function, with non-compile time constant arguments (and therefore without compile time evaluation).

constexpr

```
constexpr int get_size(int shift) {
    return sizeof(int) << shift;
}

int main() {
    int n;
    std::cin >> n;
    int x = get_size(n);
    std::cout << x;

    return 0;
}
```

In case of function it means:

1) Check that function can be evaluated in compile time if all arguments are constants

2) If check is ok, call of such function can be used (with constant arguments) where you need compile time constant

3) And where needed, it will be evaluated in comp time

Still can be used as a usual function, with non-compile time constant arguments (and therefore without compile time evaluation).

It is done so to avoid code duplication for constexpr/not constexpr functions.

constexpr

```
constexpr int get_size(int shift) {
    return sizeof(int) << shift;
}

int main() {

    int x = get_size(13);
    std::cout << x;

    return 0;
}
```

In case of function it means:

1) Check that function can be evaluated in compile time if all arguments are constants

2) If check is ok, call of such function can be used (with constant arguments) where you need compile time constant

3) And where needed, it will be evaluated in comp time

Even if you call your constexpr function from compile time constant arguments there is no guarantee that it will be evaluated in compile time!

constexpr

```cpp
constexpr int get_size(int shift) {
    return sizeof(int) << shift;
}

int main() {

    constexpr int x = get_size(13);
    std::cout << x;

    return 0;
}
```

In case of function it means:

1) Check that function can be evaluated in compile time if all arguments are constants

2) If check is ok, call of such function can be used (with constant arguments) where you need compile time constant

3) And where needed, it will be evaluated in comp time

Even if you call your constexpr function from compile time constant arguments there is no guarantee that it will be evaluated in compile time!

You need to trigger it (e.g. via constexpr when you call it).

consteval (since C++20)

```
consteval int get_size(int shift) {
    return sizeof(int) << shift;
}

int main() {

    const int x = get_size(13);
    std::cout << x;

    return 0;
}
```

In case of function it means:

1) Check that function can be evaluated in compile time if all arguments are constants

2) If check is ok, call of such function can be used (with constant arguments) where you need compile time constant

3) All uses are also checked: arguments should always be constants

4) So, it is always evaluated in compile time.

consteval (since C++20)

```
consteval int get_size(int shift) {
    return sizeof(int) << shift;
}

int main() {

    int n;
    std::cout >> n;
    constexpr int x = get_size(n);
    std::cout << x;

    return 0;
}
```

```
error: the value of 'n' is not
usable in a constant expression
```

In case of function it means:

1) Check that function can be evaluated in compile time if all arguments are constants

2) If check is ok, call of such function can be used (with constant arguments) where you need compile time constant

3) All uses are also checked: arguments should always be constants

4) So, it is always evaluated in compile time.

constexpr

```cpp
constexpr int get_size(int shift) {
    return sizeof(int) << shift;
}

int main() {
    constexpr int size = get_size(3);
    std::bitset<size> mask;
    for (size_t idx = 0; idx < size; idx++) {
        mask[idx] = (idx % 2);
    }
    std::cout << mask;

    return 0;
}
```

In case of function it means:

1) Check that function can be evaluated in compile time if all arguments are constants

But what can we use inside of such function? What will pass this check?

constexpr functions

Originally, in C++11, constexpr functions had many limitations.

Basically, you was able to:

1) Operate with (compile time) constants inside,
2) Call other constexpr functions,
3) Use static_assert
4) ...

constexpr functions

Originally, in C++11, constexpr functions had many limitations.

Basically, you was able to:

1) Operate with (compile time) constants inside,
2) Call other constexpr functions,
3) Use static_assert
4) ...

So, no loops for example, only recursion (very similar to our metaprogramming experience).

constexpr functions

Originally, in C++11, constexpr functions had many limitations.

Basically, you was able to:

1) Operate with (compile time) constants inside,
2) Call other constexpr functions,
3) Use static_assert
4) ...

So, no loops for example, only recursion (very similar to our metaprogramming experience).

But it was really long time ago. Since then constexpr evolved dramatically.

Getting prime number in compile time via metaprogramming.

```cpp
#include <iostream>
#include <type_traits>

template <int N, int M>
struct is_prime_recursion
    : std::conditional_t<(N % M != 0), is_prime_recursion<N, M - 1>,
                         std::integral_constant<bool, false>> {};

template <int N>
struct is_prime_recursion<N, 1> : std::integral_constant<bool, true> {};

template <int N> struct is_prime : is_prime_recursion<N, N - 1> {};

template <int M>
struct prime_recursion
    : std::conditional_t<is_prime<M>::value, std::integral_constant<int, M>,
                         prime_recursion<M + 1>> {};

template <int N> struct prime : prime_recursion<prime<N - 1>::value + 1> {};

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

int main() { std::cout << prime<5>::value << std::endl; }
```

Getting prime number in compile time via constexpr.

```cpp
constexpr bool is_prime(int value) {
    int i = 2;
    while (i*i <= value) {
        if (value % i == 0) {
            return false;
        }
        i++;
    }
    return true;
}

constexpr int get_nth_prime(int n) {
    int result = 2;
    while (n > 0) {
        if (is_prime(result)) {
            n--;
        }
        result++;
    }
    return result - 1;
}
```

Getting prime number in compile time via constexpr.

```
constexpr bool is_prime(int value) {
    int i = 2;
    while (i*i <= value) {
        if (value % i == 0) {
            return false;
        }
        i++;
    }
    return true;
}

constexpr int get_nth_prime(int n) {
    int result = 2;
    while (n > 0) {
        if (is_prime(result)) {
            n--;
        }
        result++;
    }
    return result - 1;
}

constexpr auto x = get_nth_prime(201); // 1229
```

Getting prime number in compile time via constexpr.

```
constexpr bool is_prime(int value) {
    int i = 2;
    while (i*i <= value) {
        if (value % i == 0) {
            return false;
        }
        i++;
    }
    return true;
}

constexpr int get_nth_prime(int n) {
    int result = 2;
    while (n > 0) {
        if (is_prime(result)) {
            n--;
        }
        result++;
    }
    return result - 1;
}

constexpr auto x = get_nth_prime(201); // 1229… and much faster than with metaprogramming!
```

```cpp
constexpr bool is_prime(int value) {
    int i = 2;
    while (i*i <= value) {
        if (value % i == 0) {
            return false;
        }
        i++;
    }
    return true;
}

constexpr int get_nth_prime(int n) {
    int result = 2;
    while (n > 0) {
        if (is_prime(result)) {
            // n--;
        }
        result++;
    }
    return result - 1;
}

constexpr auto x = get_nth_prime(201); // ???
```

```cpp
constexpr bool is_prime(int value) {
    int i = 2;
    while (i*i <= value) {
        if (value % i == 0) {
            return false;
        }
        i++;
    }
    return true;
}

constexpr int get_nth_prime(int n) {
    int result = 2;
    while (n > 0) {
        if (is_prime(result)) {
            // n--;
        }
        result++;
    }
    return result - 1;
}

constexpr auto x = get_nth_prime(201); // ???
```

```
note: constexpr evaluation hit maximum step
limit; possible infinite loop?
  36 |     while (n > 0) {

error: constexpr variable 'x' must be
initialized by a constant expression
```

constexpr functions

Originally, in C++11, constexpr functions had many limitations.

Basically, you was able to:

1) Operate with (compile time) constants inside,
2) Call other constexpr functions,
3) Use static_assert
4) ...

constexpr functions

In C++20, `constexpr` functions:

1) Can't call non-`constexpr` functions,

constexpr functions

In C++20, constexpr functions:

1) Can't call non-constexpr functions,
2) Can't return non-Literal types,

constexpr functions

In C++20, constexpr functions:

1) Can’t call non-constexpr functions,
2) Can’t return non-Literal types, can’t take arguments of non-Literal types,

constexpr functions

In C++20, constexpr functions:

1) Can't call non-constexpr functions,

2) Can't return non-Literal types, can't take arguments of non-Literal types,
3) Can't have declaration of local variable of non-Literal types,

constexpr functions

In C++20, `constexpr` functions:

1) Can’t call non-`constexpr` functions,

2) Can’t return non-`Literal` types, can’t take arguments of non-`Literal` types,
3) Can’t have declaration of local variable of non-`Literal` types,

4) Can’t have `goto` operator,
5) ...

constexpr functions

In C++23 even these limitations are weakened!

In C++20, constexpr functions:

1) Can’t call non-constexpr functions,

2) Can’t return non-Literal types, can’t take arguments of non-Literal types,
3) Can’t have declaration of local variable of non-Literal types,

4) Can’t have goto operator,
5) ...

constexpr functions

But how could throw work in compile time?

```
constexpr int get_nth_prime(int n) {
    int result = 2;
    while (n > 0) {
        if (is_prime(result)) {
            n--;
        }
        result++;
    }
    return result - 1;
}
```

constexpr functions

But how could throw work in compile time?

```
constexpr int get_nth_prime(int n) {
    if (n < 0) { throw "error"; }
    int result = 2;
    while (n > 0) {
        if (is_prime(result)) {
            n--;
        }
        result++;
    }
    return result - 1;
}
```

constexpr functions

But how could `throw` work in compile time?

this is ok

```
constexpr int get_nth_prime(int n) {
    if (n < 0) { throw "error"; }
    int result = 2;
    while (n > 0) {
        if (is_prime(result)) {
            n--;
        }
        result++;
    }
    return result - 1;
}
```

```
constexpr auto x =
        get_nth_prime(201);
```

constexpr functions

But how could `throw` work in compile time?

```
constexpr int get_nth_prime(int n) {
    if (n < 0) { throw "error"; }
    int result = 2;
    while (n > 0) {
        if (is_prime(result)) {
            n--;
        }
        result++;
    }
    return result - 1;
}
```

this is ok

```
constexpr auto x =
              get_nth_prime(201);
```

```
constexpr auto x2 =
              get_nth_prime(-201);
```

```
error: constexpr variable 'x2' must be
initialized by a constant expression
```

constexpr functions

Similar situation: change smth outside of constexpr context

```
int a;
constexpr int get_nth_prime(int n) {
    if (n < 0) { a = 13; }
    int result = 2;
    while (n > 0) {
        if (is_prime(result)) {
            n--;
        }
        result++;
    }
    return result - 1;
}
```

this is ok

```
constexpr auto x =
            get_nth_prime(201);


constexpr auto x2 =
            get_nth_prime(-201);
```

```
error: constexpr variable 'x2' must be
initialized by a constant expression
```

constexpr functions

`But how could new/delete work in compile time?`

constexpr functions

But how could new/delete work in compile time?

```
constexpr int qux(int n) {
    int* x = new int[n];
    for (size_t i = 0; i < n; i++) {
        x[i] = i * i;
    }
    int result = x[0] + x[n - 1];
    return result;
}
```

constexpr functions

But how could new/delete work in compile time?

```
constexpr int qux(int n) {
    int* x = new int[n];
    for (size_t i = 0; i < n; i++) {
        x[i] = i * i;
    }
    int result = x[0] + x[n - 1];
    return result;
}

constexpr int r = qux(10);
```

constexpr functions

But how could new/delete work in compile time?

```
constexpr int qux(int n) {
    int* x = new int[n];
    for (size_t i = 0; i < n; i++) {
        x[i] = i * i;
    }
    int result = x[0] + x[n - 1];
    return result;
}

constexpr int r = qux(10);
```

```
note: allocation performed here was not
deallocated
  61 |     int* x = new int[n];
```

```
error: constexpr variable 'r' must be
initialized by a constant expression
```

constexpr functions

But how could new/delete work in compile time?

```
constexpr int qux(int n) {
    int* x = new int[n];
    for (size_t i = 0; i < n; i++) {
        x[i] = i * i;
    }
    int result = x[0] + x[n - 1];
    delete[] x;
    return result;
}

constexpr int r = qux(10);
```

Everything works!✨

constexpr functions

But how could new/delete work in compile time?

```
constexpr int qux(int n) {
    int* x = new int[n];
    for (size_t i = 0; i < n; i++) {
        x[i] = i * i;
    }
    int result = x[0] + x[n - 1];
    delete[] x;
    return result;
}

constexpr int r = qux(10);
```

Such allocation is called transient memory allocation.

Everything works!✨

So, constexpr evaluation engine has its own "sanitizer" that checks such stuff.

constexpr functions

So, in `constexpr` functions you can do almost whatever you want (with some not so critical limitations).

constexpr functions

So, in `constexpr` functions you can do almost whatever you want (with some not so critical limitations).

However, there are (were) some limitations with `LiteralTypes`.

constexpr functions

So, in constexpr functions you can do almost whatever you want (with some not so critical limitations).

However, there are (were) some limitations with LiteralTypes.

Can we somehow work not only with primitive LiteralTypes, but with our own custom structs and classes?

constexpr functions

So, in constexpr functions you can do almost whatever you want (with some not so critical limitations).

However, there are (were) some limitations with LiteralTypes.

Can we somehow work not only with primitive LiteralTypes, but with our own custom structs and classes?

OF COURSE.

user-defined literals

```
struct Point {
    const int limit = 1024;
    int x, y;
    Point(int x, int y): x(x), y(y) {
        if (x > limit || y > limit) {
            throw std::runtime_error("too far away");
        }
    }
};

constexpr Point getPoint(int value) {
    return Point(value, value);
}
```

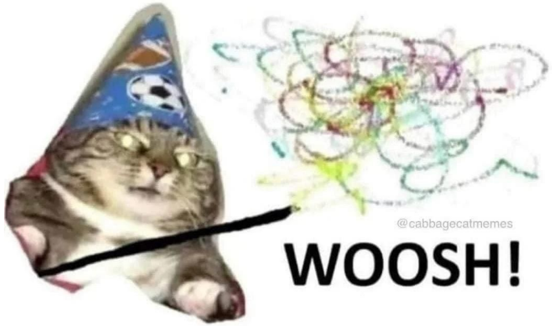
user-defined literals

```
struct Point {
    const int limit = 1024;
    int x, y;
    Point(int x, int y): x(x), y(y) {
        if (x > limit || y > limit) {
            throw std::runtime_error("too far away");
        }
    }
};

constexpr Point getPoint(int value) {
    return Point(value, value);
}
```

```
error: constexpr function's
return type 'Point' is not a
literal type
```

user-defined literals




```
struct Point {
    const int limit = 1024;
    int x, y;
    constexpr Point(int x, int y): x(x), y(y) {
        if (x > limit || y > limit) {
            throw std::runtime_error("too far away");
        }
    }
};

constexpr Point getPoint(int value) {
    return Point(value, value);
}
```

Now it is ok! Point is literal type!

```cpp
struct Point {
    const int limit = 1024;
    int x, y;
    constexpr Point(int x, int y): x(x), y(y) {
        if (x > limit || y > limit) {
            throw std::runtime_error("too far away");
        }
    }
};

constexpr Point getPoint(int value) {
    return Point(value, value);
}


int main() {
    constexpr Point p = getPoint(10);
    std::cout << p.x << " " << p.y << std::endl;
}
```

```cpp
struct Point {
    const int limit = 1024;
    int x, y;
    constexpr Point(int x, int y): x(x), y(y)
        if (x > limit || y > limit) {
            throw std::runtime_error("too far
        }
    }
};

constexpr Point getPoint(int value) {
    return Point(value, value);
}


int main() {
    constexpr Point p = getPoint(10);
    std::cout << p.x << " " << p.y << std::en
}
```

```asm
.LC0:
        .string " "
main:
        push    rbp
        mov     rbp, rsp
        mov     eax, DWORD PTR p.0[rip+4]
        mov     esi, eax
        mov     edi, OFFSET FLAT:std::cout
        call    std::basic_ostream<char, std::char_traits
        mov     esi, OFFSET FLAT:.LC0
        mov     rdi, rax
        call    std::basic_ostream<char, std::char_traits
        mov     rdx, rax
        mov     eax, DWORD PTR p.0[rip+8]
        mov     esi, eax
        mov     rdi, rdx
        call    std::basic_ostream<char, std::char_traits
        mov     esi, OFFSET FLAT:std::basic_ostream<char,
        mov     rdi, rax
        call    std::basic_ostream<char, std::char_traits
        mov     eax, 0
        pop     rbp
        ret
p.0:
        .long   1024
        .long   10
        .long   10
```

https://godbolt.org/z/ThfYGdsfP

```
struct Point {
    const int limit = 1024;
    int x, y;
    constexpr Point(int x, int y): x(x), y(y) {
        if (x > limit || y > limit) {
            throw std::runtime_error("too far away");
        }
    }
    constexpr Point operator+(const Point& other) {
        return {x + other.x, y + other.y};
    }

};

int main() {
    constexpr Point p{10, 2};
    std::cout << p.x << " " << p.y << std::endl;
}
```

```cpp
struct Point {
    const int limit = 1024;
    int x, y;
    constexpr Point(int x, int y): x(x), y(y) {
        if (x > limit || y > limit) {
            throw std::runtime_error("too far away");
        }
    }
    constexpr Point operator+(const Point& other) {
        return {x + other.x, y + other.y};
    }
};

constexpr Point operator "" _x(unsigned long long x) {
    return Point{(int) x, 0};
}
```

```cpp
struct Point {
    const int limit = 1024;
    int x, y;
    constexpr Point(int x, int y): x(x), y(y) {
        if (x > limit || y > limit) {
            throw std::runtime_error("too far away");
        }
    }
    constexpr Point operator+(const Point& other) {
        return {x + other.x, y + other.y};
    }
};

constexpr Point operator "" _x(unsigned long long x) {
    return Point{(int) x, 0};
}
```

After this you can write: `123_x` and it will give you a Point (123,0)

```cpp
struct Point {
    const int limit = 1024;
    int x, y;
    constexpr Point(int x, int y): x(x), y(y) {
        if (x > limit || y > limit) {
            throw std::runtime_error("too far away");
        }
    }
    constexpr Point operator+(const Point& other) {
        return {x + other.x, y + other.y};
    }
};

constexpr Point operator "" _x(unsigned long long x) {
    return Point{(int) x, 0};
}

constexpr Point operator "" _y(unsigned long long y) {
    return Point{0, (int) y};
}
```

```cpp
struct Point {
    const int limit = 1024;
    int x, y;
    constexpr Point(int x, int y): x(x), y(y) {
        if (x > limit || y > limit) {
            throw std::runtime_error("too far away");
        }
    }
    constexpr Point operator+(const Point& other) {
        return {x + other.x, y + other.y};
    }
};

constexpr Point operator "" _x(unsigned long long x) {
    return Point{(int) x, 0};
}

constexpr Point operator "" _y(unsigned long long y) {
    return Point{0, (int) y};
}

constexpr Point p = 10_x + 14_y;
```

```cpp
struct Point {
    const int limit = 1024;
    int x, y;
    constexpr Point(int x, int y): x(x), y(y) {
        if (x > limit || y > limit) {
            throw std::runtime_error("too far away");
        }
    }
    constexpr Point operator+(const Point& other) {
        return {x + other.x, y + other.y};
    }
};

constexpr Point operator "" _x(unsigned long long x) {
    return Point{(int) x, 0};
}

constexpr Point operator "" _y(unsigned long long y) {
    return Point{0, (int) y};
}

constexpr Point p = 10_x + 14_y;
```

```
p.0:
        .long   1024
        .long   10
        .long   14
```

https://godbolt.org/z/Ke9h1x8Ms

Compile time requirements

Where compile time constants are needed in C++?

```
template <typename T, int SIZE>
class LimitedArray {
    T* values;
public:
    LimitedArray(): values(new T[SIZE]) {}
    ~LimitedArray() {delete[] values; }
    T& operator[] (const size_t idx) {
        if (idx > SIZE) {
            throw std::out_of_range();
        }
        return values[idx];
    }
};
```

```cpp
template <typename T, int SIZE>
class ConstSequenceArray {
    T values[SIZE];
public:
    constexpr ConstSequenceArray() {
        for (auto i = 0; i < SIZE; i++) {
            values[i] = i;
        }
    }
    constexpr const T& operator[] (const size_t idx) const  {
        if (idx > SIZE) {
            throw std::out_of_range("out of range");
        }
        return values[idx];
    }
};
```

```
template <typename T, int SIZE>
class ConstSequenceArray {
    T values[SIZE];
public:
    constexpr ConstSequenceArray() {
        for (auto i = 0; i < SIZE; i++) {
            values[i] = i;
        }
    }
    constexpr const T& operator[] (const size_t idx) const  {
        if (idx > SIZE) {
            throw std::out_of_range("out of range");
        }
        return values[idx];
    }
};


constexpr ConstSequenceArray<int, 64> arr{};
```

```cpp
template <typename T, int SIZE>
class ConstSequenceArray {
    T values[SIZE];
public:
    constexpr ConstSequenceArray() {
        for (auto i = 0; i < SIZE; i++) {
            values[i] = i;
        }
    }
    constexpr const T& operator[] (const size_t idx) {
        if (idx > SIZE) {
            throw std::out_of_range("out of range");
        }
        return values[idx];
    }
};


constexpr ConstSequenceArray<int, 64> arr{};
```

```asm
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 152
        mov     DWORD PTR [rbp-272], 0
        mov     DWORD PTR [rbp-268], 1
        mov     DWORD PTR [rbp-264], 2
        mov     DWORD PTR [rbp-260], 3
        mov     DWORD PTR [rbp-256], 4
        mov     DWORD PTR [rbp-252], 5
        mov     DWORD PTR [rbp-248], 6
        mov     DWORD PTR [rbp-244], 7
        mov     DWORD PTR [rbp-240], 8
        mov     DWORD PTR [rbp-236], 9
        mov     DWORD PTR [rbp-232], 10
        mov     DWORD PTR [rbp-228], 11
        mov     DWORD PTR [rbp-224], 12
        mov     DWORD PTR [rbp-220], 13
        mov     DWORD PTR [rbp-216], 14
        mov     DWORD PTR [rbp-212], 15
        mov     DWORD PTR [rbp-208], 16
        mov     DWORD PTR [rbp-204], 17
        mov     DWORD PTR [rbp-200], 18
        mov     DWORD PTR [rbp-196], 19
        mov     DWORD PTR [rbp-192], 20
        mov     DWORD PTR [rbp-188], 21
        mov     DWORD PTR [rbp-184], 22
        mov     DWORD PTR [rbp-180], 23
        mov     DWORD PTR [rbp-176], 24
```

https://godbolt.org/z/7h85nacq4

```cpp
template <typename T, int SIZE>
class ConstSequenceArray {
    T values[SIZE];
public:
    constexpr ConstSequenceArray() {
        for (auto i = 0; i < SIZE; i++) {
            values[i] = i;
        }
    }
    constexpr const T& operator[] (const size_t idx) const  {
        if (idx > SIZE) {
            throw std::out_of_range("out of range");
        }
        return values[idx];
    }
};


constexpr ConstSequenceArray<int, 64> arr{};
constexpr auto val = arr[14];
```

Can we somehow generalize it? Make it work not only as readonly data?

```cpp
template <typename T, int SIZE>
class SequenceArray {
    T values[SIZE];
public:
    constexpr SequenceArray() {
        for (auto i = 0; i < SIZE; i++) {
            values[i] = i;
        }
    }

    constexpr const T& operator[] (const size_t idx) const {
        if (idx > SIZE) { throw std::out_of_range("out of range"); }
        return values[idx];
    }

};
```

```cpp
template <typename T, int SIZE>
class SequenceArray {
    T values[SIZE];
public:
    constexpr SequenceArray() {
        for (auto i = 0; i < SIZE; i++) {
            values[i] = i;
        }
    }

    constexpr const T& operator[] (const size_t idx) const {
        if (idx > SIZE) { throw std::out_of_range("out of range"); }
        return values[idx];
    }

    T& operator[](const size_t idx) {
        if (idx > SIZE) { throw std::out_of_range("out of range"); }
        return values[idx];
    }
};
```

```cpp
template <typename T, int SIZE>
class SequenceArray {
    T values[SIZE];
public:
    constexpr SequenceArray() {
        for (auto i = 0; i < SIZE; i++) {
            values[i] = i;
        }
    }

    constexpr const T& operator[] (const size_t idx) const {
        if (idx > SIZE) { throw std::out_of_range("out of range"); }
        return values[idx];
    }

    T& operator[](const size_t idx) {
        if (idx > SIZE) { throw std::out_of_range("out of range"); }
        return values[idx];
    }
};
```

You can’t overload functions only by return values.

But const modifier helps here!

```cpp
template <typename T, int SIZE>
class SequenceArray {
    T values[SIZE];
public:
    constexpr SequenceArray() {
        for (auto i = 0; i < SIZE; i++) {
            values[i] = i;
        }
    }

    constexpr const T& operator[] (const size_t idx) const {
        if (idx > SIZE) { throw std::out_of_range("out of range"); }
        return values[idx];
    }

    T& operator[](const size_t idx) {
        if (idx > SIZE) { throw std::out_of_range("out of range"); }
        return values[idx];
    }
};
```

```cpp
constexpr SequenceArray<int, 64> arr{};
constexpr auto element = arr[34];

SequenceArray<int, 12> arr2{};
arr2[5] = 10;
```

Getting prime number in compile time via constexpr.

```
constexpr bool is_prime(int value) {
    int i = 2;
    while (i*i <= value) {
        if (value % i == 0) {
            return false;
        }
        i++;
    }
    return true;
}

constexpr int get_nth_prime(int n) {
    int result = 2;
    while (n > 0) {
        if (is_prime(result)) {
            n--;
        }
        result++;
    }
    return result - 1;
}

constexpr auto x = get_nth_prime(201); // 1229… and much faster than with metaprogramming!
```

Getting prime number in compile time via constexpr.

What if we want to compute first N prime numbers in compile-time?

```
constexpr bool is_prime(int value) {
    int i = 2;
    while (i*i <= value) {
        if (value % i == 0) {
            return false;
        }
        i++;
    }
    return true;
}

constexpr int get_nth_prime(int n) {
    int result = 2;
    while (n > 0) {
        if (is_prime(result)) {
            n--;
        }
        result++;
    }
    return result - 1;
}

constexpr auto x = get_nth_prime(201); // 1229… and much faster than with metaprogramming!
```

```cpp
template <typename T, int SIZE>
class SequenceArray {
    T values[SIZE];
public:
    constexpr SequenceArray() {
        for (auto i = 0; i < SIZE; i++) {
            values[i] = i;
        }
    }

    constexpr const T& operator[] (const size_t idx) const {
        if (idx > SIZE) { throw std::out_of_range("out of range"); }
        return values[idx];
    }

    T& operator[](const size_t idx) {
        if (idx > SIZE) { throw std::out_of_range("out of range"); }
        return values[idx];
    }
};
```

```cpp
template <typename T, int SIZE>
class PrimeNumberArray {
    T values[SIZE];
public:
    constexpr PrimeNumberArray() {
        for (auto i = 0; i < SIZE; i++) {
            values[i] = get_nth_prime(i);
        }
    }

    constexpr const T& operator[] (const size_t idx) const {
        if (idx > SIZE) { throw std::out_of_range("out of range"); }
        return values[idx];
    }

    T& operator[](const size_t idx) {
        if (idx > SIZE) { throw std::out_of_range("out of range"); }
        return values[idx];
    }
};
```

How to fight copy-paste?

```
template <typename T, int SIZE>
class PrimeNumberArray {
    T values[SIZE];
public:
    constexpr PrimeNumberArray() {
        for (auto i = 0; i < SIZE; i++) {
            values[i] = get_nth_prime(i);
        }
    }

    constexpr const T& operator[] (const size_t idx) const {
        if (idx > SIZE) { throw std::out_of_range("out of range"); }
        return values[idx];
    }

    T& operator[](const size_t idx) {
        if (idx > SIZE) { throw std::out_of_range("out of range"); }
        return values[idx];
    }
};
```

```cpp
template <typename T, int SIZE>
class Array {
    T values[SIZE];
public:
    constexpr Array() {
        for (auto i = 0; i < SIZE; i++) {
            values[i] = 0;
        }
    }
    constexpr const T& operator[](const size_t idx) const {
        if (idx > SIZE) {
            throw std::out_of_range("out of range");
        }
        return values[idx];
    }
    T& operator[](const size_t idx) {
        if (idx > SIZE) {
            throw std::out_of_range("out of range");
        }
        return values[idx];
    }
};
```

```
template <typename T, int SIZE, typename Initializer>
class Array {
    T values[SIZE];
public:
    constexpr Array(Initializer init) {
        for (auto i = 0; i < SIZE; i++) {
            values[i] = init(i);
        }
    }
    constexpr const T& operator[](const size_t idx) const {
        if (idx > SIZE) {
            throw std::out_of_range("out of range");
        }
        return values[idx];
    }
    T& operator[](const size_t idx) {
        if (idx > SIZE) {
            throw std::out_of_range("out of range");
        }
        return values[idx];
    }
};
```

initializer type argument
to init elements

```cpp
template <typename T, int SIZE, typename Initializer>
class Array {
    T values[SIZE];
public:
    constexpr Array(Initializer init) {
        for (auto i = 0; i < SIZE; i++) {
            values[i] = init(i);
        }
    }
    ...
};


template <int SIZE>
class PrimArray {
    Array<int, SIZE, int(size_t)> array;
public:

};
```

initializer type argument
to init elements

```cpp
template <typename T, int SIZE, typename Initializer>
class Array {
    T values[SIZE];
public:
    constexpr Array(Initializer init) {
        for (auto i = 0; i < SIZE; i++) {
            values[i] = init(i);
        }
    }
    ...
};
```

initializer type argument to init elements

```cpp
template <int SIZE>
class PrimArray {
    Array<int, SIZE, int(size_t)> array;
public:
    constexpr PrimArray(): array([](size_t idx) { return get_nth_prime(idx); }) { }
};
```

```cpp
template <typename T, int SIZE, typename Initializer>
class Array {
    T values[SIZE];
public:
    constexpr Array(Initializer init) {
        for (auto i = 0; i < SIZE; i++) {
            values[i] = init(i);
        }
    }
    ...
};
```

initializer type argument to init elements

```cpp
template <int SIZE>
class PrimArray {
    Array<int, SIZE, int(size_t)> array;
public:
    constexpr PrimArray(): array([](size_t idx) { return get_nth_prime(idx); }) { }
};

constexpr PrimArray<199> arr{};
```

this will work! 🎉

```
template <typename T, int SIZE, typename Initializer>
class Array {
    T values[SIZE];
public:
    constexpr Array(Initializer init) {
        for (auto i = 0; i < SIZE; i++) {
            values[i] = init(i);
        }
    }
    ...
};
```

initializer type argument to init elements

```
template <int SIZE>
class PrimArray {
    Array<int, SIZE, int(size_t)> array;
public:
    constexpr PrimArray(): array([](size_t idx) { return get_nth_prime(idx); }) { }
};

constexpr PrimArray<199> arr{};
```

Since C++17 lambdas also can be constexpr!

this will work! 🎉

Takeaways on constexpr

- `constexpr` is modern way for compile-time execution (`metaprogramming` is also cool, but this one is usually much more convenient)

Takeaways on constexpr

- constexpr is modern way for compile-time execution (metaprogramming is also cool, but this one is usually much more convenient)

- started as quite simple feature, but improved to almost full support of C++ features

Takeaways on constexpr

- constexpr is modern way for compile-time execution (metaprogramming is also cool, but this one is usually much more convenient)

- started as quite simple feature, but improved to almost full support of C++ features

- constexpr ALL the things!
  https://www.youtube.com/watch?v=PJwd4JLYJJY

  (well, not all, but it is very widely spread across std)

Where else `constexpr` are used?

constexpr-if

```
template<typename N>
void foo(N n) {
   if (std::is_copy_constructible_v<N>) {
       N local(n);
   } else {
       N local;
       local.initialize(n);
   }
}
```

constexpr-if

```
template<typename N>
void foo(N n) {
   if (std::is_copy_constructible_v<N>) {
       N local(n);
   } else {
       N local;
       local.initialize(n);
   }
}

struct Point {
   int x, y;
   Point(): x(0), y(0) {}
   Point(int x, int y): x(x), y(y) {}
   Point(const Point& other) = delete;
   void initialize(const Point& other){ … }
};
```

constexpr-if

```
template<typename N>
void foo(N n) {
   if (std::is_copy_constructible_v<N>) {
       N local(n);
   } else {
       N local;
       local.initialize(n);
   }
}

struct Point {
   int x, y;
   Point(): x(0), y(0) {}
   Point(int x, int y): x(x), y(y) {}
   Point(const Point& other) = delete;
   void initialize(const Point& other){ … }
};
```

```
foo(Point{1 ,2});
```

constexpr-if

```
template<typename N>
void foo(N n) {
   if (std::is_copy_constructible_v<N>) {
       N local(n);
   } else {
       N local;
       local.initialize(n);
   }
}

struct Point {
   int x, y;
   Point(): x(0), y(0) {}
   Point(int x, int y): x(x), y(y) {}
   Point(const Point& other) = delete;
   void initialize(const Point& other){ … }
};
```

```
foo(Point{1 ,2});
```

```
error: use of deleted function
'Point::Point(const Point&)'
```

constexpr-if

```
template<typename N>
void foo(N n) {
   if constexpr (std::is_copy_constructible_v<N>) {
       N local(n);
   } else {
       N local;
       local.initialize(n);
   }
}

struct Point {
   int x, y;
   Point(): x(0), y(0) {}
   Point(int x, int y): x(x), y(y) {}
   Point(const Point& other) = delete;
   void initialize(const Point& other){ … }
};
```

constexpr-if

must be constant expression

```
template<typename N>
void foo(N n) {
   if constexpr (std::is_copy_constructible_v<N>) {
       N local(n);
   } else {
       N local;
       local.initialize(n);
   }
}

struct Point {
   int x, y;
   Point(): x(0), y(0) {}
   Point(int x, int y): x(x), y(y) {}
   Point(const Point& other) = delete;
   void initialize(const Point& other){ … }
};
```

constexpr-if

```
template<typename N>
void foo(N n) {
   if constexpr (std::is_copy_constructible_v<N>) {
       N local(n);
   } else {
       N local;
       local.initialize(n);
   }
}

struct Point {
   int x, y;
   Point(): x(0), y(0) {}
   Point(int x, int y): x(x), y(y) {}
   Point(const Point& other) = delete;
   void initialize(const Point& other){ … }
};
```

must be constant expression

unsuitable branch will be just eliminated! So, no compilation error.

Variadic templates

```
template<typename None = void>
void foo() {}


template<typename Head, typename... Tail>
void foo(Head head, Tail... tail) {
   if (sizeof...(tail) == 0) {
       std::cout << "Well, looks like " << head << " is the last one." << std::endl;
   } else {
       std::cout << "Looking at element " << head << ", " << std::endl;
       foo(tail...);
   }
}
```

Variadic templates

We need this ugly specialization because we can’t handle zero length list properly (we still need to somehow call foo(tail…))

```
template<typename None = void>
void foo() {}

template<typename Head, typename... Tail>
void foo(Head head, Tail... tail) {
    if (sizeof...(tail) == 0) {
        std::cout << "Well, looks like " << head << " is the last one." << std::endl;
    } else {
        std::cout << "Looking at element " << head << ", " << std::endl;
        foo(tail...);
    }
}
```

Variadic templates

Not anymore! Now we will just remove (do not compile) the second branch if tail is empty.

```
template<typename Head, typename... Tail>
void foo(Head head, Tail... tail) {
    if constexpr (sizeof...(tail) == 0) {
        std::cout << "Well, looks like " << head << " is the last one." << std::endl;
    } else {
        std::cout << "Looking at element " << head << ", " << std::endl;
        foo(tail...);
    }
}
```

Constraints

```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;
}

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

copy(dist, source);
```

```
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);
```

Thanks to SFINAE this will work, but looks more like a workaround

```cpp
template <typename T>
T abs(T x) {
   return x >= 0 ? x : -x;
}

template<typename T, std::enable_if_t<std::is_floating_point<T>::value, int> = 0>
bool equals(T a, T b) {
   const double epsilon = 0.000001;
   return abs(a - b) < static_cast<T>(epsilon);
}

template<typename T, std::enable_if_t<!std::is_floating_point<T>::value, int> = 0>
bool equals(T a, T b) {
   return a == b;
}

int main() {
   std::cout << equals(0.5, 0.50000000001) << std::endl;
   std::cout << equals(5, 5) << std::endl;
   return 0;
}
```

The same situation here: we specialize our template on the whole family of types.

```
template <typename T>
T abs(T x) {
   return x >= 0 ? x : -x;
}

template<typename T, std::enable_if_t<std::is_floating_point<T>::value, int> = 0>
bool equals(T a, T b) {
   const double epsilon = 0.000001;
   return abs(a - b) < static_cast<T>(epsilon);
}

template<typename T, std::enable_if_t<!std::is_floating_point<T>::value, int> = 0>
bool equals(T a, T b) {
   return a == b;
}

int main() {
   std::cout << equals(0.5, 0.50000000001) << std::endl;
   std::cout << equals(5, 5) << std::endl;
   return 0;
}
```

```cpp
template <typename T>
T abs(T x) {
   return x >= 0 ? x : -x;
}

template<typename T, std::enable_if_t<std::is_floating_point<T>::value, int> = 0>
bool equals(T a, T b) {
   const double epsilon = 0.000001;
   return abs(a - b) < static_cast<T>(epsilon);
}

template<typename T, std::enable_if_t<!std::is_floating_point<T>::value, int> = 0>
bool equals(T a, T b) {
   return a == b;
}

int main() {
   std::cout << equals(0.5, 0.50000000001) << std::endl;
   std::cout << equals(5, 5) << std::endl;
   return 0;
}
```

```cpp
template <typename T>
T abs(T x) {
    return x >= 0 ? x : -x;
}

template<typename T>
bool equals(T a, T b) requires(std::is_floating_point<T>::value) {
    const double epsilon = 0.000001;
    return abs(a - b) < static_cast<T>(epsilon);
}

template<typename T>
bool equals(T a, T b) requires(!std::is_floating_point<T>::value) {
    return a == b;
}

int main() {
    std::cout << equals(0.5, 0.50000000001) << std::endl;
    std::cout << equals(5, 5) << std::endl;
    return 0;
}
```

With help of requires you can define constraints on your template arguments.

```cpp
template <typename T>
T abs(T x) {
   return x >= 0 ? x : -x;
}

template<typename T>
bool equals(T a, T b) requires(std::is_floating_point<T>::value) {
   const double epsilon = 0.000001;
   return abs(a - b) < static_cast<T>(epsilon);
}

template<typename T>
bool equals(T a, T b) requires(!std::is_floating_point<T>::value) {
   return a == b;
}

int main() {
   std::cout << equals(0.5, 0.50000000001) << std::endl;
   std::cout << equals(5, 5) << std::endl;
   return 0;
}
```

With help of requires you can define constraints on your template arguments.

Inside: any constexpr.

```
template <typename T>
T abs(T x) {
    return x >= 0 ? x : -x;
}

template<typename T>
bool equals(T a, T b) requires(std::is_floating_point<T>::value) {
    const double epsilon = 0.000001;
    return abs(a - b) < static_cast<T>(epsilon);
}

template<typename T>
bool equals(T a, T b) requires(!std::is_floating_point<T>::value) {
    return a == b;
}

int main() {
    std::cout << equals(0.5, 0.50000000001) << std::endl;
    std::cout << equals(5, 5) << std::endl;
    return 0;
}
```

```
constexpr bool is_power_of_two(int n) {
   return (n & (n - 1)) == 0;
}
```

With help of requires you can define constraints on your template arguments.

Inside: any constexpr.

```cpp
constexpr bool is_power_of_two(int n) {
    return (n & (n - 1)) == 0;
}

template<typename T, int initial_capacity>
requires(is_power_of_two(initial_capacity))
class HashMap {
    T* array;
public:
    HashMap(): array(new T[initial_capacity]) {}
};
```

With help of requires you can define constraints on your template arguments.

Inside: any constexpr.

```cpp
constexpr bool is_power_of_two(int n) {
    return (n & (n - 1)) == 0;
}

template<typename T, int initial_capacity>
requires(is_power_of_two(initial_capacity))
class HashMap {
    T* array;
public:
    HashMap(): array(new T[initial_capacity]) {}
};


HashMap<int, 8> map;  // ok
```

With help of requires you can define constraints on your template arguments.

Inside: any constexpr.

```
constexpr bool is_power_of_two(int n) {
    return (n & (n - 1)) == 0;
}

template<typename T, int initial_capacity>
requires(is_power_of_two(initial_capacity))
class HashMap {
    T* array;
public:
    HashMap(): array(new T[initial_capacity]) {}
};


HashMap<int, 8> map;  // ok
HashMap<int, 6> map2;
```

With help of requires you can define constraints on your template arguments.

Inside: any constexpr.

```cpp
constexpr bool is_power_of_two(int n) {
    return (n & (n - 1)) == 0;
}

template<typename T, int initial_capacity>
requires(is_power_of_two(initial_capacity))
class HashMap {
    T* array;
public:
    HashMap(): array(new T[initial_capacity]) {}
};


HashMap<int, 8> map;  // ok
HashMap<int, 6> map2;
```

With help of requires you can define constraints on your template arguments.

Inside: any constexpr.

```
error: template constraint failure for
'template<class T, int initial_capacity>
requires  is_power_of_two()(initial_capacity)
class HashMap'
```

```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;
}

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

copy(dist, source);
```

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

template <typename T>
T* copy(T* dist, T* source) requires(!std::is_trivially_copy_assignable_v<T>) {
   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);
```

Requires

`Actually requires is much more powerful.`

Requires

Actually requires is much more powerful.

```
template<typename T>
void my_sort(T begin, T end) {
    std::sort(begin, end);
}
```

Requires

Actually requires is much more powerful.

```
template<typename T>
void my_sort(T begin, T end) {
    std::sort(begin, end);
}

template<typename T>
struct Box { T val; };

std::vector<Box<int>> v = {{13}, {42}, {53}};
my_sort(v.begin(), v.end());
```

```
E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/bits/predefined_ops.h: In instantiation of 'constexpr bool __gnu_cxx::__ops::_Iter_less_iter::operator()(_Iterator1, _Iterator2) const [with _Iterator1 = __gnu_cxx::__normal_iterator<Box<int>*, std::vector<Box<int> > >; _Iterator2 = __gnu_cxx::__normal_iterator<Box<int>*, std::vector<Box<int> > >]':
E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/bits/stl_algo.h:1826:14:   required from 'constexpr void std::__insertion_sort(_RandomAccessIterator, _RandomAccessIterator, _Compare) [with _RandomAccessIterator = __gnu_cxx::__normal_iterator<Box<int>*, std::vector<Box<int> > >; _Compare = __gnu_cxx::__ops::_Iter_less_iter]'
E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/bits/stl_algo.h:1866:25:   required from 'constexpr void std::__final_insertion_sort(_RandomAccessIterator, _RandomAccessIterator, _Compare) [with _RandomAccessIterator = __gnu_cxx::__normal_iterator<Box<int>*, std::vector<Box<int> > >; _Compare = __gnu_cxx::__ops::_Iter_less_iter]'
E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/bits/stl_algo.h:1957:31:   required from 'constexpr void std::__sort(_RandomAccessIterator, _RandomAccessIterator, _Compare) [with _RandomAccessIterator = __gnu_cxx::__normal_iterator<Box<int>*, std::vector<Box<int> > >; _Compare = __gnu_cxx::__ops::_Iter_less_iter]'
E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/bits/stl_algo.h:4842:18:   required from 'constexpr void std::sort(_RAIter, _RAIter) [with _RAIter = __gnu_cxx::__normal_iterator<Box<int>*, std::vector<Box<int> > >]'
E:/CLionProjects/untitled1/main.cpp:1514:14:   required from 'void my_sort(T, T) [with T = __gnu_cxx::__normal_iterator<Box<int>*, std::vector<Box<int> > >]'
E:/CLionProjects/untitled1/main.cpp:1525:12:   required from here
E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/bits/predefined_ops.h:45:23: error: no match for 'operator<' (operand types are 'Box<int>' and 'Box<int>')
   45 |       { return *__it1 < *__it2; }
      |                ~~~~~~~^~~~~~~~
In file included from E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/bits/stl_algobase.h:67,
                 from E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/bits/char_traits.h:39,
                 from E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/ios:40,
                 from E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/ostream:38,
                 from E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/iostream:39,
                 from E:/CLionProjects/untitled1/main.cpp:1:
E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/bits/stl_iterator.h:1112:5: note: candidate: 'template<class _IteratorL, class _IteratorR, class _Container> constexpr std::__detail::__synth3way_t<_IteratorR, _IteratorL> __gnu_cxx::operator<=>(const __gnu_cxx::__normal_iterator<_IteratorL, _Container>&, const __gnu_cxx::__normal_iterator<_IteratorR, _Container>&)' (reversed)
 1112 |     operator<=>(const __normal_iterator<_IteratorL, _Container>& __lhs,
      |     ^~~~~~~~
E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/bits/stl_iterator.h:1112:5: note:   template argument deduction/substitution failed:
In file included from E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/bits/stl_algobase.h:71,
                 from E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/bits/char_traits.h:39,
                 from E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/ios:40,
                 from E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/ostream:38,
                 from E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/iostream:39,
                 from E:/CLionProjects/untitled1/main.cpp:1:
E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/bits/predefined_ops.h:45:23: note:   'Box<int>' is not derived from 'const __gnu_cxx::__normal_iterator<_IteratorL, _Container>'
   45 |       { return *__it1 < *__it2; }
      |                ~~~~~~~^~~~~~~~
E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/bits/predefined_ops.h: In instantiation of 'constexpr bool __gnu_cxx::__ops::_Val_less_iter::operator()(_Value&, _Iterator) const [with _Value = Box<int>; _Iterator = __gnu_cxx::__normal_iterator<Box<int>*, std::vector<Box<int> > >]':
E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/bits/stl_algo.h:1806:20:   required from 'constexpr void std::__unguarded_linear_insert(_RandomAccessIterator, _Compare) [with _RandomAccessIterator = __gnu_cxx::__normal_iterator<Box<int>*, std::vector<Box<int> > >; _Compare = __gnu_cxx::__ops::_Val_less_iter]'
E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/bits/stl_algo.h:1834:36:   required from 'constexpr void std::__insertion_sort(_RandomAccessIterator, _RandomAccessIterator, _Compare) [with _RandomAccessIterator = __gnu_cxx::__normal_iterator<Box<int>*, std::vector<Box<int> > >; _Compare = __gnu_cxx::__ops::_Iter_less_iter]'
E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/bits/stl_algo.h:1866:25:   required from 'constexpr void std::__final_insertion_sort(_RandomAccessIterator, _RandomAccessIterator, _Compare) [with _RandomAccessIterator = __gnu_cxx::__normal_iterator<Box<int>*, std::vector<Box<int> > >; _Compare = __gnu_cxx::__ops::_Iter_less_iter]'
E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/bits/stl_algo.h:1957:31:   required from 'constexpr void std::__sort(_RandomAccessIterator, _RandomAccessIterator, _Compare) [with _RandomAccessIterator = __gnu_cxx::__normal_iterator<Box<int>*, std::vector<Box<int> > >; _Compare = __gnu_cxx::__ops::_Iter_less_iter]'
E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/bits/stl_algo.h:4842:18:   required from 'constexpr void std::sort(_RAIter, _RAIter) [with _RAIter = __gnu_cxx::__normal_iterator<Box<int>*, std::vector<Box<int> > >]'
```

```
E:/CLionProjects/untitled1/main.cpp:1514:14:   required from 'void my_sort(T, T) [with T = __gnu_cxx::__normal_iterator<Box<int>*, std::vector<Box<int> > >]'
E:/CLionProjects/untitled1/main.cpp:1525:12:   required from here
E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/bits/predefined_ops.h:98:22: error: no match for 'operator<' (operand types are 'Box<int>' and 'Box<int>')
   98 |       { return __val < *__it; }
      |                ~~~~~~^~~~~~~
In file included from E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/bits/stl_algobase.h:67,
                 from E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/bits/char_traits.h:39,
                 from E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/ios:40,
                 from E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/ostream:38,
                 from E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/iostream:39,
                 from E:/CLionProjects/untitled1/main.cpp:1:
E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/bits/stl_iterator.h:1112:5: note: candidate: 'template<class _IteratorL, class _IteratorR, class _Container> constexpr
std::__detail::__synth3way_t<_IteratorR, _IteratorL> __gnu_cxx::operator<=>(const __gnu_cxx::__normal_iterator<_IteratorL, _Container>&, const __gnu_cxx::__normal_iterator<_IteratorR, _Container>&)' (reversed)
 1112 |     operator<=>(const __normal_iterator<_IteratorL, _Container>& __lhs,
      |     ^~~~~~~~
E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/bits/stl_iterator.h:1112:5: note:   template argument deduction/substitution failed:
In file included from E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/bits/stl_algobase.h:71,
                 from E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/bits/char_traits.h:39,
                 from E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/ios:40,
                 from E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/ostream:38,
                 from E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/iostream:39,
                 from E:/CLionProjects/untitled1/main.cpp:1:
E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/bits/predefined_ops.h:98:22: note:   'Box<int>' is not derived from 'const __gnu_cxx::__normal_iterator<_IteratorL, _Container>'
   98 |       { return __val < *__it; }
      |                ~~~~~~^~~~~~~
In file included from E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/bits/char_traits.h:39,
                 from E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/ios:40,
                 from E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/ostream:38,
                 from E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/iostream:39,
                 from E:/CLionProjects/untitled1/main.cpp:1:
E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/bits/stl_algobase.h: In instantiation of 'constexpr void std::iter_swap(_ForwardIterator1, _ForwardIterator2) [with _ForwardIterator1 =
__gnu_cxx::__normal_iterator<Box<int>*, std::vector<Box<int> > >; _ForwardIterator2 = __gnu_cxx::__normal_iterator<Box<int>*, std::vector<Box<int> > >]':
E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/bits/stl_algo.h:85:20:   required from 'constexpr void std::__move_median_to_first(_Iterator, _Iterator, _Iterator, _Iterator, _Compare) [with
_Iterator = __gnu_cxx::__normal_iterator<Box<int>*, std::vector<Box<int> > >; _Compare = __gnu_cxx::__ops::_Iter_less_iter]'
E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/bits/stl_algo.h:1904:34:   required from 'constexpr _RandomAccessIterator std::__unguarded_partition_pivot(_RandomAccessIterator,
_RandomAccessIterator, _Compare) [with _RandomAccessIterator = __gnu_cxx::__normal_iterator<Box<int>*, std::vector<Box<int> > >; _Compare = __gnu_cxx::__ops::_Iter_less_iter]'
E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/bits/stl_algo.h:1938:38:   required from 'constexpr void std::__introsort_loop(_RandomAccessIterator, _RandomAccessIterator, _Size,
_Compare) [with _RandomAccessIterator = __gnu_cxx::__normal_iterator<Box<int>*, std::vector<Box<int> > >; _Size = long long int; _Compare = __gnu_cxx::__ops::_Iter_less_iter]'
E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/bits/stl_algo.h:1954:25:   required from 'constexpr void std::__sort(_RandomAccessIterator, _RandomAccessIterator, _Compare) [with
_RandomAccessIterator = __gnu_cxx::__normal_iterator<Box<int>*, std::vector<Box<int> > >; _Compare = __gnu_cxx::__ops::_Iter_less_iter]'
E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/bits/stl_algo.h:4842:18:   required from 'constexpr void std::sort(_RAIter, _RAIter) [with _RAIter = __gnu_cxx::__normal_iterator<Box<int>*,
std::vector<Box<int> > >]'
E:/CLionProjects/untitled1/main.cpp:1514:14:   required from 'void my_sort(T, T) [with T = __gnu_cxx::__normal_iterator<Box<int>*, std::vector<Box<int> > >]'
E:/CLionProjects/untitled1/main.cpp:1525:12:   required from here
E:/JetBrains/CLion 2021.3.4/bin/mingw/lib/gcc/x86_64-w64-mingw32/11.2.0/include/c++/bits/stl_algobase.h:182:11: error: call of overloaded 'swap(Box<int>&, Box<int>&)' is ambiguous
  182 |       swap(*__a, *__b);
      |       ~~~~^~~~~~~~~~~~
```

Requires

Actually requires is much more powerful.

```
template<typename T>
void my_sort(T begin, T end) {
   std::sort(begin, end);
}

template<typename T>
struct Box { T val; };

std::vector<Box<int>> v = {{13}, {42}, {53}};
my_sort(v.begin(), v.end());
```

Sometimes template arguments are too generic. It would be nice to have some constraints.

Requires

Actually requires is much more powerful.

```
template<typename T>
void my_sort(T begin, T end) {
   std::sort(begin, end);
}

template<typename T>
struct Box { T val; };

std::vector<Box<int>> v = {{13}, {42}, {53}};
my_sort(v.begin(), v.end());
```

Sometimes template arguments are too generic. It would be nice to have some constraints.

At least for better diagnostics.

Requires

Actually requires is much more powerful.

```
template<typename T>
void my_sort(T begin, T end) requires ??? {
   std::sort(begin, end);
}

template<typename T>
struct Box { T val; };

std::vector<Box<int>> v = {{13}, {42}, {53}};
my_sort(v.begin(), v.end());
```

Requires

Actually requires is much more powerful.

```
template<typename T>
void my_sort(T begin, T end) requires requires (T it1, T it2) { *it1 < *it2; } {
    std::sort(begin, end);
}

template<typename T>
struct Box { T val; };

std::vector<Box<int>> v = {{13}, {42}, {53}};
my_sort(v.begin(), v.end());
```

This expression just checks if "*it1 < *it2" is correct code.

Requires

Actually requires is much more powerful.

```
template<typename T>
void my_sort(T begin, T end) requires requires (T it1, T it2) { *it1 < *it2; } {
    std::sort(begin, end);
}

template<typename T>
struct Box { T val; };

std::vector<Box<int>> v = {{13}, {42}, {53}};
my_sort(v.begin(), v.end());
```

Requires

This expression just checks if "*it1 < *it2" is correct code.
It doesn’t try to execute it (neither in run- nor in compile- time).

Actually requires is much more powerful.

```
template<typename T>
void my_sort(T begin, T end) requires requires (T it1, T it2) { *it1 < *it2; } {
    std::sort(begin, end);
}

template<typename T>
struct Box { T val; };

std::vector<Box<int>> v = {{13}, {42}, {53}};
my_sort(v.begin(), v.end());
```

Requires

Actually requires is much more powerful.

This expression just checks if "*it1 < *it2" is correct code.
It doesn’t try to execute it (neither in run- nor in compile- time). The result is boolean (checked by 1st requires).

```
template<typename T>
void my_sort(T begin, T end) requires requires (T it1, T it2) { *it1 < *it2; } {
   std::sort(begin, end);
}

template<typename T>
struct Box { T val; };

std::vector<Box<int>> v = {{13}, {42}, {53}};
my_sort(v.begin(), v.end());
```

```
E:/CLionProjects/untitled1/main.cpp: In function 'int main()':
E:/CLionProjects/untitled1/main.cpp:1520:12: error: no matching function for call to 'my_sort(std::vector<Box<int> >::iterator, std::vector<Box<int> >::iterator)'
 1520 |     my_sort(v.begin(), v.end());
      |     ~~~~~~~^~~~~~~~~~~~~~~~~~~~
E:/CLionProjects/untitled1/main.cpp:1506:6: note: candidate: 'template<class T> void my_sort(T, T) requires requires(T it1, T it2) {*it1 < *it2;}'
 1506 | void my_sort(T begin, T end) requires requires (T it1, T it2) { *it1 < *it2; } {
      |      ^~~~~~~
E:/CLionProjects/untitled1/main.cpp:1506:6: note:   template argument deduction/substitution failed:
E:/CLionProjects/untitled1/main.cpp:1506:6: note: constraints not satisfied
E:/CLionProjects/untitled1/main.cpp: In substitution of 'template<class T> void my_sort(T, T) requires requires(T it1, T it2) {*it1 < *it2;} [with T =
__gnu_cxx::__normal_iterator<Box<int>*, std::vector<Box<int> > >]':
E:/CLionProjects/untitled1/main.cpp:1520:12:   required from here
E:/CLionProjects/untitled1/main.cpp:1506:6:   required by the constraints of 'template<class T> void my_sort(T, T) requires requires(T it1, T it2) {*it1 < *it2;}'
E:/CLionProjects/untitled1/main.cpp:1506:39:   in requirements with 'T it1', 'T it2' [with T = __gnu_cxx::__normal_iterator<Box<int>*, std::vector<Box<int>, std::allocator<Box<int>
> > >]
E:/CLionProjects/untitled1/main.cpp:1506:70: note: the required expression '((* it1) < (* it2))' is invalid
 1506 | void my_sort(T begin, T end) requires requires (T it1, T it2) { *it1 < *it2; } {
      |                                                                 ~~~~~^~~~~~
cc1plus.exe: note: set '-fconcepts-diagnostics-depth=' to at least 2 for more detail
ninja: build stopped: subcommand failed.
```

Diagnostic is much better, this is already a great advantage.

Requires

This expression just checks if "*it1 < *it2" is correct code.
It doesn't try to execute it (neither in run- nor in compile- time). The result is boolean (checked by 1st requires).

Actually requires is much more powerful.

```
template<typename T>
void my_sort(T begin, T end) requires requires (T it1, T it2) { *it1 < *it2; } {
   std::sort(begin, end);
}

template<typename T>
struct Box { T val; };

std::vector<int> v = {13, 42, 53};
my_sort(v.begin(), v.end());
```

This will work, no constraints violations.

```
template<typename T>
void my_sort(T begin, T end) requires requires (T it1, T it2) { *it1 < *it2; } {
   std::sort(begin, end);
}

template<typename T>
struct Box { T val; };

std::vector<int> v = {13, 42, 53};
my_sort(v.begin(), v.end());
```

```cpp
template<typename T>
concept IterToComparable = requires(T it1, T it2) { *it1 < *it2; };

template<typename T>
void my_sort(T begin, T end) requires requires (T it1, T it2) { *it1 < *it2; } {
   std::sort(begin, end);
}

template<typename T>
struct Box { T val; };

std::vector<int> v = {13, 42, 53};
my_sort(v.begin(), v.end());
```

```
template<typename T>
concept IterToComparable = requires(T it1, T it2) { *it1 < *it2; };

template<IterToComparable T>
void my_sort(T begin, T end) {
   std::sort(begin, end);
}

template<typename T>
struct Box { T val; };

std::vector<int> v = {13, 42, 53};
my_sort(v.begin(), v.end());
```

Concept - is just a named set of such constraints.
Can be used instead of typename.

```
template<typename T>
concept IterToComparable = requires(T it1, T it2) { *it1 < *it2; };

template<IterToComparable T>
void my_sort(T begin, T end) {
   std::sort(begin, end);
}

template<typename T>
struct Box { T val; };

std::vector<int> v = {13, 42, 53};
my_sort(v.begin(), v.end());
```

```cpp
template<typename T>
concept Printable = requires(T val) {
    std::cout << val;
};
```

```cpp
template<typename T>
concept Printable = requires(T val) {
   std::cout << val;
};

template<typename T>
concept Iterable = requires(T collection) {
   collection.begin();
   collection.end();
};
```

```
template<typename T>
concept Printable = requires(T val) {
   std::cout << val;
};

template<typename T>
concept Iterable = requires(T collection) {
   collection.begin();
   collection.end();
};
```

Actually, much more to check here, but let it be so.

```cpp
template<typename T>
concept Printable = requires(T val) {
   std::cout << val;
};

template<typename T>
concept Iterable = requires(T collection) {
   collection.begin();
   collection.end();
};
```

Actually, much more to check here, but let it be so.

```cpp
template<typename T>
concept IterableNotPrintable = Iterable<T> && !Printable<T>;
```

You can combine different concepts here to create new one

```cpp
template<typename T>
concept Printable = requires(T val) {
   std::cout << val;
};

template<typename T>
concept Iterable = requires(T collection) {
   collection.begin();
   collection.end();
};

template<typename T>
concept IterableNotPrintable = Iterable<T> && !Printable<T>;

template<Printable T> void print(const T& p) {
   std::cout << p;
}
```

Actually, much more to check here, but let it be so.

```cpp
template<typename T>
concept Printable = requires(T val) { std::cout << val; };

template<typename T>
concept Iterable = requires(T collection) {
   collection.begin();
   collection.end();
};

template<typename T>
concept IterableNotPrintable = Iterable<T> && !Printable<T>;

template<Printable T> void print(const T& p) { std::cout << p; }

template<IterableNotPrintable T> void print(const T& collection) {
   for (auto&& e: collection) { std::cout << e << ", "; }
}

template<typename T> void print(const T& val) {
   std::cout << "don't now how to print val" << std::endl;
}
```

Concepts

You can combine: constexpr (including requires), SFINAE conditions and other concepts via && or ||.

```
template<typename T>
concept SmallIntegral = std::is_integral_v<T> && sizeof(T) < 4;
```

Concepts

You can combine: constexpr (including requires), SFINAE conditions and other concepts via && or ||.

```
template<typename T>
concept SmallIntegral = std::is_integral_v<T> && sizeof(T) < 4;


template<SmallIntegral T>
void copy(T* src, T* dst) {
   std::cout << "copy small integral";
}


template<typename T>
void copy(T* src, T* dst) requires(std::is_integral_v<T> && sizeof(T) >= 4) {
   std::cout << "copy usual integral";
}
```

Concepts

You can combine: constexpr (including requires), SFINAE conditions and other concepts via && or ||.

```
template<typename T>
concept SmallIntegral = std::is_integral_v<T> && sizeof(T) < 4;


template<typename T>
concept BoxOfSmallIntegral = requires(T box) {
   { box.value } -> SmallIntegral;
};
```

Concepts

You can combine: constexpr (including requires), SFINAE conditions and other concepts via && or ||.

```
template<typename T>
concept SmallIntegral = std::is_integral_v<T> && sizeof(T) < 4;


template<typename T>
concept BoxOfSmallIntegral = requires(T box) {
    { box.value } -> SmallIntegral;
};
```

Here we define a concept:

1) "box.value" should compile
2) its type should be SmallIntegral

Concepts

You can combine: `constexpr` (including `requires`), SFINAE conditions and other concepts via && or ||.

```
template<typename T>
concept SmallIntegral = std::is_integral_v<T> && sizeof(T) < 4;


template<typename T>
concept BoxOfSmallIntegral = requires(T box) {
   { box.value } -> SmallIntegral;
};


template<typename T>
concept CollectionOfSmallIntegral = requires(T collection) {
   { collection[0] } -> SmallIntegral;
};
```

Concepts

You can combine: constexpr (including requires), SFINAE conditions and other concepts via && or ||.

```
template<typename T>
concept SmallIntegral = std::is_integral_v<T> && sizeof(T) < 4;


template<typename T>
concept BoxOfSmallIntegral = requires(T box) {
   { box.value } -> SmallIntegral;
};


template<typename T>
concept CollectionOfSmallIntegral = requires(T collection) {
   { collection[0] } -> SmallIntegral;
};
```

Will it work?

```cpp
template<typename T>
concept CollectionOfSmallIntegrals = requires(T collection) {
   { collection[0] } -> SmallIntegral;
};

template<CollectionOfSmallIntegrals T>
void foooo(T arr) {
   std::cout << "collection of small integrals" << std::endl;
}

template<typename T>
void foooo(T arr) {
   std::cout << "not collection of small integrals" << std::endl;
}

int main() {
   Vector<int> v1;    foooo(v1);
   Vector<char> v2;   foooo(v2);

   return 0;
}
```

```cpp
template<typename T>
concept CollectionOfSmallIntegrals = requires(T collection) {
    { collection[0] } -> SmallIntegral;
};

template<CollectionOfSmallIntegrals T>
void foooo(T arr) {
    std::cout << "collection of small integrals" << std::endl;
}

template<typename T>
void foooo(T arr) {
    std::cout << "not collection of small integrals" << std::endl;
}

int main() {
    Vector<int> v1;    foooo(v1);  // not collection of small integrals
    Vector<char> v2;   foooo(v2);

    return 0;
}
```

```cpp
template<typename T>
concept CollectionOfSmallIntegrals = requires(T collection) {
   { collection[0] } -> SmallIntegral;
};

template<CollectionOfSmallIntegrals T>
void foooo(T arr) {
   std::cout << "collection of small integrals" << std::endl;
}

template<typename T>
void foooo(T arr) {
   std::cout << "not collection of small integrals" << std::endl;
}

int main() {
   Vector<int> v1;    foooo(v1);  // not collection of small integrals
   Vector<char> v2;   foooo(v2);  // not collection of small integrals… why?

   return 0;
}
```

```cpp
template<typename T>
concept CollectionOfSmallIntegrals = requires(T collection) {
    { collection[0] } -> SmallIntegral;
};
```

this one returns reference!

```cpp
template<CollectionOfSmallIntegrals T>
void foooo(T arr) {
    std::cout << "collection of small integrals" << std::endl;
}

template<typename T>
void foooo(T arr) {
    std::cout << "not collection of small integrals" << std::endl;
}

int main() {
    Vector<int> v1;    foooo(v1);  // not collection of small integrals
    Vector<char> v2;   foooo(v2);  // not collection of small integrals… why?

    return 0;
}
```

Concepts

You can combine: `constexpr` (including `requires`), SFINAE conditions and other concepts via && or ||.

```
template<typename T>
concept SmallIntegral = std::is_integral_v<T> && sizeof(T) < 4;


template<typename T>
concept BoxOfSmallIntegral = requires(T box) {
    { box.value } -> SmallIntegral;
};
```

Will it work?

```
template<typename T>
concept CollectionOfSmallIntegral = requires(T collection) {
    { collection[0] } -> SmallIntegral;
};
```

Concepts

You can combine: constexpr (including requires), SFINAE conditions and other concepts via && or ||.

```
template<typename T>
concept SmallIntegral = std::is_integral_v<
                            std::remove_cvref_t<T>> && sizeof(T) < 4;


template<typename T>
concept BoxOfSmallIntegral = requires(T box) {
   { box.value } -> SmallIntegral;
};


template<typename T>
concept CollectionOfSmallIntegral = requires(T collection) {
   { collection[0] } -> SmallIntegral;
};
```

```cpp
template<typename T>
concept CollectionOfSmallIntegrals = requires(T collection) {
    { collection[0] } -> SmallIntegral;
};
```

this one returns reference!

```cpp
template<CollectionOfSmallIntegrals T>
void foooo(T arr) {
    std::cout << "collection of small integrals" << std::endl;
}

template<typename T>
void foooo(T arr) {
    std::cout << "not collection of small integrals" << std::endl;
}

int main() {
    Vector<int> v1;    foooo(v1);  // not collection of small integrals
    Vector<char> v2;   foooo(v2);  // collection of small integrals

    return 0;
}
```

Concepts

- Previously missing part of templates in C++: static interfaces with needed constraints

Concepts

- Previously missing part of templates in C++: static interfaces with needed constraints
- Improve diagnostics and make specialization much easier (to compare with enable_if + SFINAE)

Concepts

- Previously missing part of templates in C++: static interfaces with needed constraints

- Improve diagnostics and make specialization much easier (to compare with enable_if + SFINAE)

- Work well with constexpr and SFINAE-like checks

Concepts

- Previously missing part of templates in C++: static interfaces with needed constraints

- Improve diagnostics and make specialization much easier (to compare with enable_if + SFINAE)

- Work well with constexpr and SFINAE-like checks

- Heavily used in modern STL: ranges, iterators, ...

Not So Tiny Task №14 (1 point)

Implement a function:

```
template<size_t SIZE, typename... Types>
void allocate(void* memory, Types... args) { ... }
```

That takes some preallocated memory of the given *SIZE*, and variadic number of arguments of different types. Than it initializes new elements of the corresponding types inside the given memory as copies of the given values. Use placement new for that.

Not So Tiny Task №14 (1 point)

Implement a function:

```
template<size_t SIZE, typename... Types>
void allocate(void* memory, Types... args) { ... }
```

That takes some preallocated memory of the given SIZE, and variadic number of arguments of different types. Than it initializes new elements of the corresponding types inside the given memory as copies of the given values. Use placement new for that.

It should be statically checked that SIZE is enough for such allocation.

It should be statically checked that all types are copy_constructable.

Not So Tiny Task №15 (1 point)

Implement a container:

```
template<typename... Types>
class Container {
    ...

public:
    Container(Types... args) { … }

    template<typename T>
    T getElement(size_t idx) { … }
};
```

That encapsulates memory storage for all given arguments (placed in some memory sequentially, one by one).

Not So Tiny Task №15 (1 point)

Example:

```
Container<int, char, Point> c(12, 'c', Point{2 ,3});
std::cout << c.getElement<int>(0) << std::endl;
std::cout << c.getElement<char>(1) << std::endl;
std::cout << c.getElement<Point>(2) << std::endl;
```