



Back To Basics Templates

NICOLAI JOSUTTIS



Stop war
Stop Putin

Nicolai M. Josuttis

- **Independent consultant**
 - Continuously learning since 1962
- **C++:**
 - since 1990
 - ISO Standard Committee since 1997
- **Other Topics:**
 - Systems Architect
 - Technical Manager
 - SOA
 - X and OSF/Motif



C++

©2022 by josuttis.com

Templates

C++98

- **Generic code for arbitrary types/values**
 - Defined with `template<placeholders>`
 - Types/values become clear when the generic code is used
 - The code gets compiled (instantiated) for each specific type/value
 - Type binding still applies
- **Became a very very powerful language feature**
 - More than initially expected
 - More important than inheritance (see the standard library)
- **Primitive recursive language**
 - Far more than just text/code replacement
 - You can use templates to „compute“ at compile time

C++

©2022 by josuttis.com

3

josuttis | eckstein
IT communication

Modern C++

Function Templates

C++

©2022 by josuttis.com

4

josuttis | eckstein
IT communication

Function Templates

C++98

- Generic function code for arbitrary types

```
template<typename T>
T mymax(T a, T b)
{
    return b < a ? a : b;
}
```

Template definition

- T is a common name, but anything else is possible (and maybe more readable)
- Also `class` instead of `typename` (same meaning)

Template instantiations

- Compile code for **deduced** or **specified** types

```
int i1=42, i2=77;
std::cout << mymax(i1, i2);
```

compiles and calls:

```
int mymax(int a, int b) {
    return b < a ? a : b;
}
```

```
std::cout << mymax(0.7, 33.4);
```

compiles and calls:

```
double mymax(double a, double b) {
    return b < a ? a : b;
}
```

```
std::string s{"hi"}, t{"world"};
std::cout << mymax(s, t);
```

compiles and calls:

```
std::string mymax(std::string a, std::string b) {
    return b < a ? a : b;
}
```

```
s = mymax<std::string>("hi", "ho");
```

C++

©2022 by josuttis.com

5

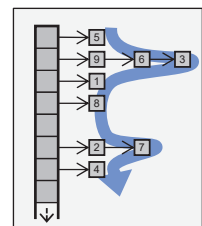
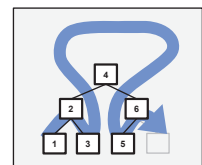
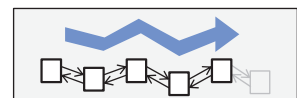
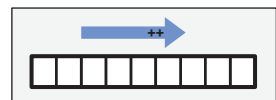
josuttis | eckstein
IT communication

Generic Iterating

C++

- Iterate over elements of different containers with the same generic code:

```
// print all elements:
for (const auto& elem : coll) {
    std::cout << elem << '\n';
}
```

**C++**

©2022 by josuttis.com

6

josuttis | eckstein
IT communication

Generic Iterating with Function Templates

C++98

- Iterate over elements of different containers with the same generic code:

```
template<typename T>
void print(const T& coll)
{
    for (const auto& elem : coll) {
        std::cout << elem << '\n';
    }
}
```

"for any type T compile"

```
std::vector<int> v;
...
print(v);
```

compiles:

```
void print(const std::vector<int>& coll)
{
    ...
}
```

```
std::set<std::string> s;
...
print(s);
```

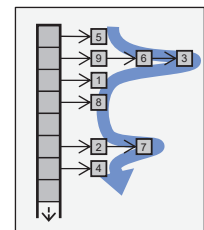
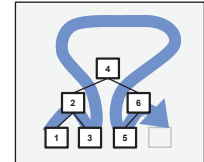
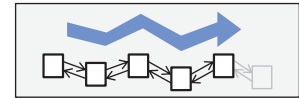
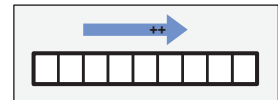
compiles:

```
void print(const std::set<std::string>& coll)
{
    ...
}
```

```
std::vector<double> v2;
...
print(v2);
```

compiles:

```
void print(const std::vector<double>& coll)
{
    ...
}
```



C++

©2022 by josuttis.com

7

josuttis | eckstein
IT communication

Templates in Header Files

C++98

- Templates are usually **defined** in header files
 - Not only *declared*
 - No `inline` necessary

mycode.hpp:

```
template<typename T>
T mymax(T a, T b)
{
    return b < a ? a : b;
}
```

#include "mycode.hpp"

...

```
int i1=42, i2=77;
auto a = mymax(i1, i2); // OK
auto b = mymax(0.7, 33.4); // OK
std::string s{"he"}, t{"ho"};
auto c = mymax(s, t); // OK
```

mycode.hpp:

```
template<typename T>
T mymax(T a, T b);
```

#include "mycode.hpp"

...

```
int i1=42, i2=77;
auto a = mymax(i1, i2); // Error
auto b = mymax(0.7, 33.4); // Error
std::string s{"he"}, t{"ho"};
auto c = mymax(s, t); // Error
```

C++

©2022 by josuttis.com

8

josuttis | eckstein
IT communication

auto Parameters for Ordinary Functions (since C++20)

C++20

```
template<typename T>
void printColl(const T& coll)
{
    for (const auto& elem : coll) {
        std::cout << elem << '\n';
    }
}
```

Abbreviated function templates:

```
void printColl(const auto& coll)
{
    for (const auto& elem : coll) {
        std::cout << elem << '\n';
    }
}
```

```
std::vector<int> v;
std::list<double> l;
...
printColl(v);
printColl(l);
printColl("hello");
printColl<std::string>("hi");
```

- Generic code
- Usually in header files
- No inline

Equivalent except that **T** is not available

C++

©2022 by josuttis.com

9

josuttis | eckstein
IT communication

Function Template Requirements

C++98

- Template require that all **operations are supported**
 - Still checking still alive

```
template<typename T>
T mymax(T a, T b)
{
    return b < a ? a : b;
}
```

Implicit requirements for T:

- operator < (returning bool)
- copy/move constructor

```
int i1=42, i2=77;
std::cout << mymax(i1, i2);
```

OK, compiles and calls:

```
int mymax(int a, int b) {
    return b < a ? a : b;
}
```

```
std::cout << mymax(7, 33.4);
```

// ERROR: can't deduce T (int or double)

```
std::cout << mymax<double>(7, 33.4);
```

// OK, T is double

```
std::complex<double> c1, c2;
```

```
std::cout << mymax(c1, c2);
```

// ERROR: deduces T as complex<>, but no < supported

```
std::atomic<int> a1{8}, a2{15};
```

```
std::cout << mymax(a1, a2);
```

// ERROR: deduces T as atomic<>, but copying disabled

C++

©2022 by josuttis.com

10

josuttis | eckstein
IT communication

Stop war
Stop Putin

Concepts

C++20

- **Concepts:** (since C++20)
 - To formulate **formal constraints** for generic code
 - To disable bad behavior or find errors early

Concept:
Named requirements

```
template<typename T>
concept HasLessThan = requires (T x) { {x < x} -> std::convertible_to<bool>; };
```

```
template<typename T>
requires std::copyable<T> && HasLessThan<T>
T mymax(T a, T b)
{
    return b < a ? a : b;
}
```

Explicit constraint for T:

- copy/move constructor
- operator < returning bool

```
int i1=42, i2=77;
std::cout << mymax(i1, i2);
```

```
std::complex<double> c1, c2;
std::cout << mymax(c1, c2); // ERROR: concept HasLessThan not supported
```

```
std::atomic<int> a1{8}, a2{15};
std::cout << mymax(a1, a2); // ERROR: concept std::copyable not supported
```

C++

©2022 by josuttis.com

11

josuttis | eckstein
IT communication

Stop war
Stop Putin

Multiple Template Parameters

C++98

- You can provide multiple template parameters

```
template<typename T1, typename T2>
void print(const T1& val1, const T2& val2)
{
    std::cout << val1 << ' ' << val2 << '\n';
}
```

• Parameters might have different types now

```
int i1=42, i2=77;
print(i1, i2); // OK, print<int, int>()
print(0.7, 33.4); // OK, print<double, double>()
print(i1, 0.7); // OK, print<int, double>()
```

```
std::string s{"hi"}, t{"world"};
print(s, t); // OK, print<std::string, std::string>()
print("hi", "world"); // OK, print<char[3], char[6]>()
print("hi", s); // OK, print<char[3], std::string>()
```

```
print<double>(i1, i2); // OK, print<double, int>()
print<double, double>(i1, i2); // OK, print<double, double>()
```

C++

©2022 by josuttis.com

12

josuttis | eckstein
IT communication

Multiple Template Parameters and the Return Type

C++98

- How to deduce/specify the return type?

```
template<typename T1, typename T2>
??? mymax(T1 a, T2 b)
{
    return b < a ? a : b;
}
```

- Parameters might have different types now

```
int i = 42;
std::string s{"hi"};
...
```

// Goal:

```
auto a1 = mymax(i, 0.7);
auto a2 = mymax(0.7, i);
```

// OK, mymax<int, double>() => double (T2)

// OK, mymax<double, int>() => double (T1)

```
auto s1 = mymax("hi", s);
auto s2 = mymax(s, "world");
```

// OK, mymax<const char*, std::string>() => std::string (T2)

// OK, mymax<std::string, const char*>() => std::string (T1)

- Return type should be the "common type" of T1 and T2 returned by operator ?:

C++

©2022 by josuttis.com

13

josuttis | eckstein
IT communication

C++14: Return Type auto

C++14

- Let the compiler deduce the return type

```
template<typename T1, typename T2>
auto mymax(T1 a, T2 b)
{
    return b < a ? a : b;
}
```

- Let the compiler deduce the return type
 - Since C++14

- Templates are usually a good application of return type **auto**
 - The type is always right
 - Better than guessing wrong

```
int i = 42;
std::string s{"hi"};
...
```

```
auto a1 = mymax(i, 0.7);
auto a2 = mymax(0.7, i);
```

// OK, return type is double

// OK, return type is double

```
auto a3 = mymax<double>(i, 0.7);
```

// OK, return type is double

```
auto a4 = mymax<long>(0.7, i);
```

// return type is long (may convert 0.7 to 0)

```
auto s1 = mymax("hi", s);
```

// OK, return type is std::string

```
auto s2 = mymax(s, "world");
```

// OK, return type is std::string

C++

©2022 by josuttis.com

14

josuttis | eckstein
IT communication

Stop war
Stop Putin

Modern C++

Class Templates

C++

©2022 by josuttis.com

15

josuttis | eckstein
IT communication

Stop war
Stop Putin

Class Templates

C++98

- Class code for arbitrary types

```
#include <vector>

template<typename T>
class Stack {
private:
    std::vector<T> elems; // elements
public:
    Stack(); // constructor
    void push(const T&); // push element
    T pop(); // pop element
    T top() const; // yield top elem
    bool empty() const { // no elements?
        return elems.empty();
    }
};
```

// stack of ints:

```
Stack<int> intStack;
intStack.push(7);
std::cout << intStack.top() << '\n';
```

// stack of strings:

```
Stack<std::string> strStack;
strStack.push("hello");
std::cout << strStack.top() << '\n';
stringStack.pop();
```

// stack of complex numbers:

```
Stack<std::complex<double>> cpxStack;
```

C++

©2022 by josuttis.com

16

josuttis | eckstein
IT communication

Implementation of Class Templates

C++98

```
#include <vector>
#include <cassert>

template<typename T>
class Stack {
private:
    std::vector<T> elems; // elements
public:
    Stack(); // constructor
    void push(const T&); // push element
    T pop(); // pop element
    T top() const; // yield top element
    bool empty() const { // no elements?
        return elems.empty();
    }
};

template<typename T>
Stack<T>::Stack()
{
    // constructor: nothing to do
}
```

```
template<typename T>
void Stack<T>::push(const T& elem)
{
    elems.push_back(elem); // insert at the end
}

template<typename T>
T Stack<T>::pop()
{
    assert(!elems.empty());
    T elem = elems.back(); // copy last element
    elems.pop_back(); // remove last element
    return elem; // returned saved copy
}

template<typename T>
T Stack<T>::top() const
{
    assert(!elems.empty());
    return elems.back(); // return last element
}
```

Function templates

- Usually in header files
- Don't need inline

C++

©2022 by josuttis.com

17

josuttis | eckstein
IT communication

Generic Member Functions are Only Instantiated if Used

C++98

- **Class Template arguments**
 - have to support all operations of member templates that are **used**
 - don't have to support all operations that **could** be used

```
template<typename T>
class Stack {
private:
    std::vector<T> elems; // elements
public:
    ...
    void print() const {
        for (const T& elem : elems) {
            std::cout << elem << ' '; // requires << for element type T
        }
    }
};
```

```
Stack<int> si; // OK
si.push(42); // OK
int i = si.top(); // OK
si.print(); // OK
```

```
Stack<std::pair<int, double>> sp; // OK
sp.push(std::pair<int, double>(1, 3.0)); // OK
sp.push({6, 7}); // OK since C++11
std::cout << sp.top().first << '\n'; // OK

sp.print(); // compile-time ERROR: << not defined for std::pair<>
```

C++

©2022 by josuttis.com

18

josuttis | eckstein
IT communication

Class Template Argument Deduction (CTAD)

C++17

- Constructors deduce templates parameters (since C++17)

```
std::complex<int> c1(5, 3);           // all C++ versions
std::complex<int> c2{5, 3};          // OK (since C++11)
std::complex c3{5, 3};               // deduces std::complex<int>
std::complex c4(5, 3);               // deduces std::complex<int>
std::complex c5 = 42;                // deduces std::complex<int>

std::vector<int> v{0, 8, 15};         // since C++11
std::vector v2{0, 8, 15};             // deduces std::vector<int>
std::vector v3{"all", "right"};       // Note: deduces vector<const char*>
std::vector v4{v.begin(), v.end()};   // Note: vector of two iterators (not the elements)
                                       // deduces vector<vector<int>::iterator>

std::vector<int> v5{v.begin(), v.end()}; // OK: copies elements from v

std::array a{0, 8, 15, 7, 4, 1, 42, 9, 5}; // deduces std::array<int, 9>
std::array a2{42, 45, 77.7};            // Error: types differ
```

Don't use CTAD unless deduction is obvious

C++

©2022 by josuttis.com

19

josuttis | eckstein
IT communication

CTAD for std::vector<>

C++17

```
namespace std {
    template <typename ElemT, typename Allocator = allocator<T>>
    class vector {
    public:
        vector() noexcept(noexcept(Allocator()));
        explicit vector(const Allocator&) noexcept;
        explicit vector(size_t n, const Allocator& = Allocator());
        vector(size_t n, const ElemT& value, const Allocator& = Allocator());
        template <typename Iter>
        vector(Iter beg, Iter end, const Allocator& = Allocator());
        vector(const vector& x);
        vector(vector&&) noexcept;
        vector(const vector&, const Allocator&);
        vector(vector&&, const Allocator&);
        vector(initializer_list<ElemT>, const Allocator& = Allocator());
        ...
    };
}
```

std::vector v1{8, 15}; // vector<int> with 2 elems

Overload resolution:

1. Number of arguments must match
2. Types must fit (incl. implicit conversions)
3. Choose best match:
 - Perfect match over template
 - Template over conversion
 - For non-empty brace initialization, std::initializer_list<> has highest priority

C++

©2022 by josuttis.com

20

josuttis | eckstein
IT communication

Stop war
Stop Putin

Modern C++

Non-Type Template Parameters

C++

©2022 by josuttis.com

21

josuttis | eckstein
IT communication

Stop war
Stop Putin

Non-Type Template Parameters (NTPP)

C++98

- Template parameters can be **values** instead of types
 - Passing different values results in different types

```
template <typename T, int Sz> // stack of at most Sz values of type T
class Stack {
private:
    T elems[Sz]; // array of Sz elements of type T
    int numElems; // actual number of elements
public:
    Stack(); // constructor
    void push(const T&); // push element into the stack
    T pop(); // pop element out of the stack
    ...
};
```

```
Stack<int, 20> int20Stack; // stack of at most 20 ints
Stack<int, 40> int40Stack; // stack of at most 40 ints
Stack<std::string, 10> stringStack; // stack of at most 10 strings
int20Stack = int40Stack; // ERROR: different types
```

C++

©2022 by josuttis.com

22

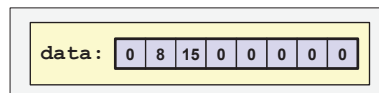
josuttis | eckstein
IT communication

Stop war
Stop Putin

std::array<>

C++11

- **std::array<ElemType, Size>**
- **Internally: Fixed-sized array**
 - All elements in **contiguous memory**
 - No external memory for the elements
 - "Vector on the stack"
 - **Random access**
 - Can jump back and forth
 - Has index operator **[]**
 - No insertion/deletion of elements



```
std::array<int, 8> a = {0, 8, 15}; // array of 8 initialized int's (other values 0)

if (!a.empty()) {
    a.back() = 99999; // modify last element
}

for (int i = 0; i < a.size(); ++i) { // modify all elements
    a[i] += 1; // as usual no check whether index is valid
}

std::sort(a.begin(), a.end()); // sort elements
```

C++

©2022 by josuttis.com

23

josuttis | eckstein
IT communication

Stop war
Stop Putin

std::array<> is a Templified Aggregate

C++11

```
struct CArray {
    int elems[10];
};
```

```
template<typename T, size_t SZ>
struct array {
    T elems[SZ];

    size_t size() const {
        return SZ;
    }

    T& operator[] (size_t idx) {
        return elems[idx];
    }
    const T& operator[] (size_t idx) const {
        return elems[idx];
    }

    typedef T* iterator;
    T* begin() {
        return &elems[0];
    }
    T* end() {
        return &elems[0] + SZ;
    }
    ...
};
```

```
struct CArray ca = {1, 2, 3, 4};

for (int i=0; i<10; ++i) {
    std::cout << ca.elems[i] << '\n';
}

for (int* p=ca.elems; p<ca.elems+10; ++p) {
    std::cout << *p << '\n';
}
```

```
array<int,10> a = {1, 2, 3, 4};
std::cout << a.size() << " elements\n";

for (size_t i=0; i<a.size(); ++i) {
    std::cout << a[i] << '\n';
}

for (array<int,10>::iterator p = a.begin();
     p != a.end(); ++p) {
    std::cout << *p << '\n';
}

for (auto p=a.begin(); p != a.end(); ++p) {
    std::cout << *p << '\n';
}

for (const auto& elem : a) {
    std::cout << elem << '\n';
}
```

C++

©2022 by josuttis.com

24

josuttis | eckstein
IT communication

Stop war
Stop Putin

Non-Type Template Parameter (NTTP) Types

C++98 / C++20

- **Supported types:**
 - Types for **constant integral values** (`int`, `long`, `enum`, ...)
 - `std::nullptr_t` (the type of `nullptr`)
 - Pointers to globally visible objects/functions/members
 - Lvalue references to objects or functions
- **Not supported are:**
 - String literals (directly)
 - Classes
- **Since C++20 supported are:**
 - Floating-point types (`float`, `double`, ...)
 - Data structures with public members
 - Lambdas

C++

©2022 by josuttis.com

25

josuttis | eckstein
IT communication

Stop war
Stop Putin

Modern C++

Variadic Templates

C++

©2022 by josuttis.com

26

josuttis | eckstein
IT communication

Variadic Templates

C++11

- Templates for a variable number of template arguments
 - Type-safe *varargs* interface
- For functions and classes
- Named *parameter packs*
 - represent multiple arguments (types/objects)
 - can be passed together to somewhere else

```
template<typename T, typename... Types>
void print(T firstArg, Types... args)
{
    std::cout << firstArg << '\n'; // output first argument
    print(args...);                // call print() for all other arguments
}
```

any number of types

any number of arguments (of any type)

C++

©2022 by josuttis.com

27

josuttis | eckstein
IT communication

Variadic Templates

C++11

```
std::string str = "world";
print( "hello", 7.5, str );

=> print<const char*, ...>( "hello", 7.5, str )
    std::cout << "hello" << '\n';
    print( 7.5, str );

=> print<double, ...>( 7.5, str )
    std::cout << 7.5 << '\n';
    print( str );

=> print<std::string>( str, )
    std::cout << str << '\n';
    print( );

void print()
{
}
```

Types represents 2 types
args represents 2 params

Types represents 1 type
args represents 1 param

Types represents 0 types
args represents 0 params

```
template<typename T, typename... Types>
void print(T firstArg, Types... args)
{
    std::cout << firstArg << '\n'; // output first argument
    print(args...);                // call print() for all other arguments
}
```

C++

©2022 by josuttis.com

28

josuttis | eckstein
IT communication

Stop war
Stop Putin

Variadic Templates

C++11

```
std::string str = "world";
print( "hello", 7.5, str );
```

Types represents 2 types
args represents 2 params

```
=> print<const char*, ...>( "hello", 7.5, str )
    std::cout << "hello" << '\n';
    print( 7.5, str );
```

Types represents 1 type
args represents 1 param

```
=> print<double, ...>( 7.5, str )
    std::cout << 7.5 << '\n';
    print( str );
```

Types represents 0 types
args represents 0 params

```
=> print<std::string>( str, )
    std::cout << str << '\n';
    print( );
```

```
void print()
{
}
```

```
template<typename T, typename... Types>
void print(T firstArg, Types... args)
{
    std::cout << firstArg << '\n';
    print( args... );
}
```

Code effectively compiled:

```
std::string str = "world";
std::cout << "hello" << '\n';
std::cout << 7.5 << '\n';
std::cout << str << '\n';
```

C++

©2022 by josuttis.com

29

josuttis | eckstein
IT communication

Stop war
Stop Putin

Parameter Packs and sizeof...

C++11

- Variadic arguments are called "*parameter packs*"
- The number of elements in a parameter pack can be 0
- Operator **sizeof...** yields the number of elements in a parameter pack
- Parameter packs can also have **qualifiers**

```
void print()
{
}
```

"template parameter pack"

```
template<typename T, typename... Types>
void print(const T& firstArg, const Types&... args)
{
    std::cout << firstArg << '\n';
    print( args... );
```

"function/call parameter pack"

```
    std::cout << sizeof...(Types) << '\n'; // print number of arguments of Types
    std::cout << sizeof...(args) << '\n'; // print number of arguments of args
}
```

C++

©2022 by josuttis.com

30

josuttis | eckstein
IT communication

Dealing with Variadic Templates

C++11 / C++17

- Note that the following code does **not compile**:

```
template<typename T, typename... Types>
void print(T firstArg, Types... args)
{
    std::cout << firstArg << '\n';
    if (sizeof...(args) > 0) { // if args is not empty
        print(args...);      // ERROR if no print() for no arguments declared
    }
}
```

Runtime if:

- Needs valid statement even if never called

- Since C++17:

```
template<typename T, typename... Types>
void print(T firstArg, Types... args)
{
    std::cout << firstArg << '\n';
    if constexpr(sizeof...(args) > 0) { // if args is not empty
        print(args...);                // OK: if no print() for no arguments declared
    }
}
```

Compile-time if:

- If condition is false valid syntax is enough

C++

©2022 by josuttis.com

31

josuttis | eckstein
IT communication

Modern C++

Bringing it all together

C++

©2022 by josuttis.com

32

josuttis | eckstein
IT communication

Stop war
Stop Putin

Let's Add Values to a Collection

C++98

```
template<typename Coll, typename T>
void add(Coll& coll, const T& val)
{
    coll.push_back(val);
}
```

```
std::vector<int> coll;
```

```
add(coll, 42);    // OK
```

C++

©2022 by josuttis.com

33

josuttis | eckstein
IT communication

Stop war
Stop Putin

auto as Function Parameters

C++20

```
void add(auto& coll, const auto& val)
{
    coll.push_back(val);
}
```

```
std::vector<int> coll;
```

```
add(coll, 42);    // OK
```

"Abbreviated function template"

- Generic code
- Equivalent to:

```
template<typename T1, typename T2>
void add(T1& coll, const T2& val) {
    coll.push_back(val);
}
```
- Definition usually in header files
- No `inline` necessary

C++

©2022 by josuttis.com

34

josuttis | eckstein
IT communication

Stop war
Stop Putin

auto as Function Parameters

C++20

```
void add(auto& coll, const auto& val)
{
    coll.push_back(val);
}

void add(auto& coll, const auto& val)
{
    coll.insert(val);
}

std::vector<int> coll1;
std::set<int> coll2;

add(coll1, 42);    // ERROR: ambiguous
add(coll2, 42);    // ERROR: ambiguous
```

C++

©2022 by josuttis.com

35

josuttis | eckstein
IT communication

Stop war
Stop Putin

Concepts as Type Constraints

C++20

```
template <typename Coll>
concept HasPushBack = requires (Coll c, Coll::value_type v) {
    c.push_back(v);
};
```

```
void add(HasPushBack auto& coll, const auto& val)
{
    coll.push_back(val);
}

void add(auto& coll, const auto& val)
{
    coll.insert(val);
}
```

```
std::vector<int> coll1;
std::set<int> coll2;
```

```
add(coll1, 42);    // OK, uses 1st add() calling push_back()
add(coll2, 42);    // OK, uses 2nd add() calling insert()
```

Equivalent to:

```
template<HasPushBack T1, typename T2>
void add(T1& coll, const T2& val) {
    coll.push_back(val);
}
```

Overload resolution prefers
more specialized template

C++

©2022 by josuttis.com

36

josuttis | eckstein
IT communication

Stop war
Stop Putin

requires and Compile-Time if

C++20

```
void add(auto& coll, const auto& val)
{
    if constexpr (requires { coll.push_back(val); }) { // if push_back() is supported
        coll.push_back(val);                          // - call push_back()
    }
    else {
        coll.insert(val);                             // - else call insert()
    }
}

std::vector<int> coll1;
std::set<int> coll2;

add(coll1, 42);   // OK, calls push_back()
add(coll2, 42);   // OK, calls insert()
```

C++

©2022 by josuttis.com

37

josuttis | eckstein
IT communication

Stop war
Stop Putin

Thank You!



Nicolai M. Josuttis

www.josuttis.com
nico@josuttis.com
@NicoJosuttis



C++

©2022 by josuttis.com

38

josuttis | eckstein
IT communication