Programming with Concepts

Petter Holmberg – C++ Stockholm 0x25 – January 2023

My Meetup Talks

Previous Talks:

- **2017 - *From Type to Concept***
- **2018 - *The Dark Art of Type Functions***
- **2019 - *Ancient Math / Modern C++***
- **2022 - *Functional Parsing in C++20***

This Time:

How to think about and use standard C++20 concepts in practice

Not included:

- **C++20 concepts and constraints syntax**
- **How the standard library concepts are implemented**
- **How to implement your own concepts**
- **Technical details on how concepts and constrains work together with templates**

Terminology

Concept:

A named set of type requirements:

- ***Syntactic*** ***(what must the type's public interface include)***
- ***Semantic*** ***(how does the type work)***
- ***Contractual*** ***(what are the pre- and post-conditions on the public interface)***
- ***Complexity*** ***(what are the performance guarantees on the type's operations)***

Concepts may also express requirements of relationships between multiple types.

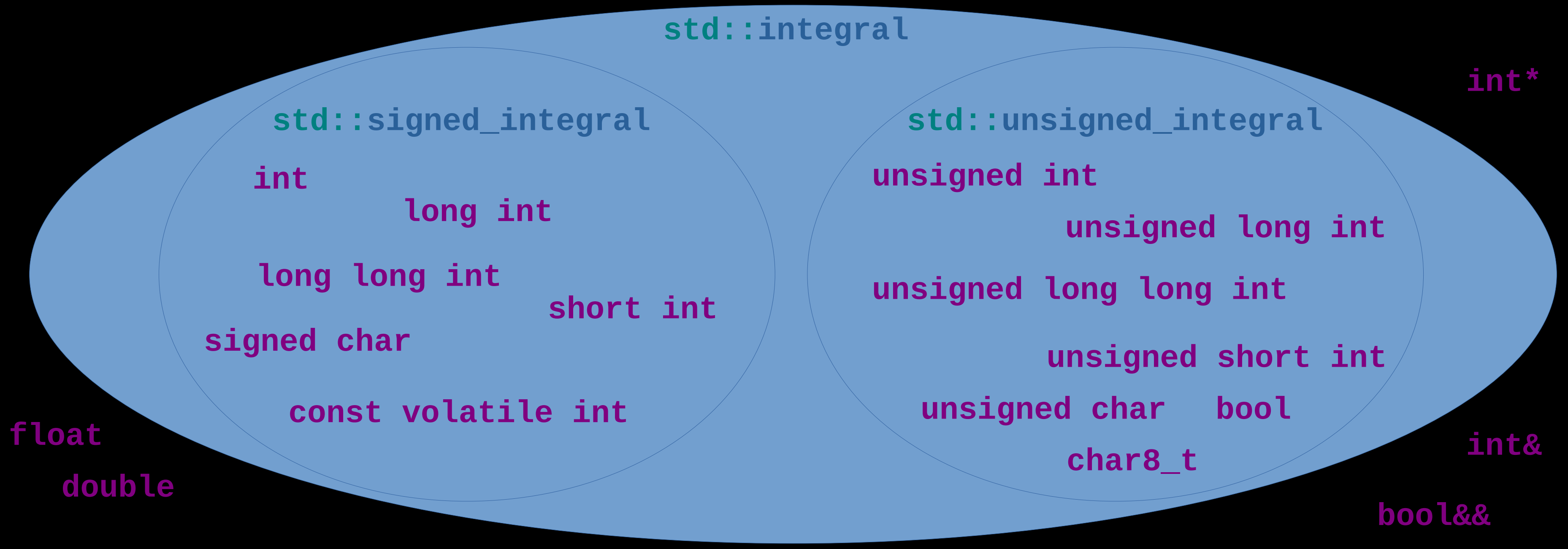
Model:

A type (or type combination) that fulfills all of the type requirements *(satisfies a concept).*

Constraint:

A requirement on template arguments that is used to select the most appropriate overload/specialization at compile time *(often a concept or combination of concepts).*

Example: Integral Types



Examples:

`int` **models** `std::integral` ***(because it satisfies the concept's requirements)***

`int` **also models** `std::signed_integral`

`int` **does not model** `std::unsigned_integral`

Using Concepts

When you *use* a C++ concept, think of it as calling a compile-time function that takes one or more *types* as input and returns a `bool`:

- **If it returns `false`, your type(s) does not model the concept.**
- **If it returns `true`, your type(s) interface fulfills the *syntactic* requirements of the concept.**

C++ concepts are implemented by stating syntactic requirements, which can be checked by the compiler.

C++ concepts cannot save you from errors due to bugs in a type's use or implementation!

The C++ concepts in the standard define the other kinds of requirements in writing *(and so should you!)*

Better Error Messages, Faster Compile Times

```
// Version 1, no constraints
template <typename T>
T compute(T a, T b)
{
    /*
    lots of code
    */
    return a + b;
}
```

Inability to add T:s would be caught here

```
// Version 2, constrained by concept
template <std::integral T>
T compute(T a, T b)
{
    /*
    lots of code
    */
    return a + b;
}
```

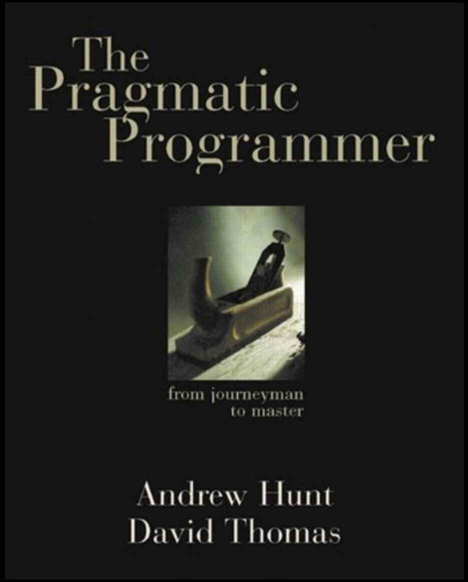
Same error would be caught here

Nothing to prevent us from doing something invalid with T here

Why Do We Write Templates?

The DRY Principle:

“Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.”

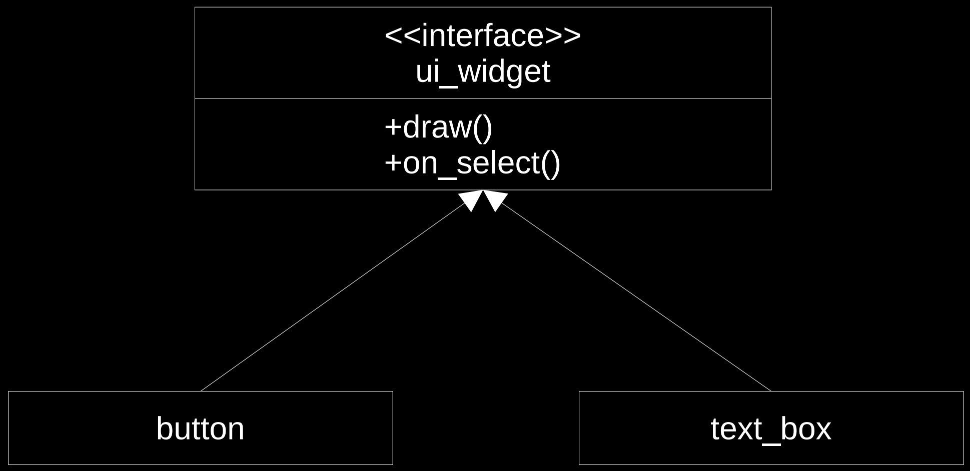


Not just about code duplication, also documentation, database schemas etc.

[The Pragmatic Programmer, 1st ed, §11 – DRY–Don’t Repeat Yourself, pp 30-34]

Concepts vs. OOP

```cpp
class ui_widget
{
public:
    virtual ~ui_widget() = default;
    virtual void draw() const = 0;
    virtual void on_select() = 0;
};

class button : public ui_widget
{
public:
    void draw() const override;
    void on_select() override;
};

class text_box : public ui_widget
{
public:
    void draw() const override;
    void on_select() override;
};
```



Concepts vs. OOP

```
class ui_widget
{
public:
    virtual ~ui_widget() = default;
    virtual void draw() const = 0;
    virtual void on_select() = 0;
};

class button : public ui_widget
{
public:
    void draw() const override;
    void on_select() override;
};

class text_box : public ui_widget
{
public:
    void draw() const override;
    void on_select() override;
};
```

Class must explicitly derive from interface

Member functions are virtual, require vtable lookup

Only member functions are overrideable

Concepts vs. OOP

```
template <typename T>
concept ui_widget = /* impl */;


class button
{
public:
    void draw() const;
    void on_select();
};

class text_box
{
public:
    void draw() const;
    void on_click();
};


static_assert(ui_widget<button>);
static_assert(ui_widget<text_box>);
```

Implicitly satisfies the ui_widget concept

No virtual functions needed, zero overhead

No common type conversions with button

Compile-time checking (no code generated)

<concepts> Library

Language: `same_as`, `derived_from`, `convertible_to`, `common_reference_with`, `common_with`

Arithmetic: `integral`, `signed_integral`, `unsigned_integral`, `floating_point`

Initialization: `assignable_from`, `swappable`, `destructible`, `constructible_from`, `default_initializable`, `move_constructible`, `copy_constructible`

Comparison: `equality_comparable`, `equality_comparable_with`, `totally_ordered`, `totally_ordered_with`

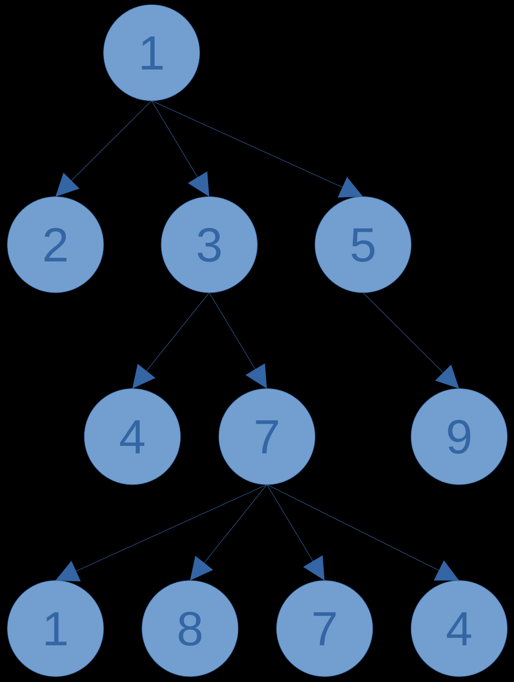
Object: `movable`, `copyable`, `semiregular`, `regular`

Callable: `invocable`, `regular_invocable`, `predicate`, `relation`, `equivalence_relation`, `strict_weak_order`

[ISO/IEC 14882:2020, §18 – Concepts Library]

Concepts and Containers

```
template <typename T>
class my_vector
{
    T* data;
    std::size_t size;
    /* impl */
};


struct int_tree
{
    int root;
    my_vector<int_tree> subtrees;
};
```



Concepts and Containers

```
template <std::copyable T>
class my_vector
{
    T* data;
    std::size_t size;
    /* impl */
};



struct int_tree
{
    int root;
    my_vector<int_tree> subtrees;
};
```

Oops! Compiler wants to check if int_tree is std::copyable

Concepts and Containers

```
template <typename T>
class my_vector
{
    T* data;
    std::size_t size;
    /* impl */
};



struct int_tree
{
    int root;
    my_vector<int_tree> subtrees;
};
```

Ok! int_tree is incomplete at this point...

…but now we can determine int_tree's layout!

```
static_assert(std::copyable<std::vector<std::unique_ptr<int>>>);
```

Compiles (surprise!)

Concepts and Containers

```
template <typename T>
class my_vector
{
    T* data;
    std::size_t size;
public:

    my_vector(my_vector const& other)
    {
        static_assert(std::copyable<T>, "Cannot implement copy ctor, T is not copyable!");
        /* impl */
    }

    /* impl */
};
```

Not a constraint but catches errors earlier in implementation!

Not Just About Templates!

```
void redraw_selected_button()
{
    auto it = std::ranges::find_if(buttons, is_selected);

    if (it != std::end(buttons)) (*it).draw();
};
```

Not Just About Templates!

```
void redraw_selected_button()
{
    std::forward_iterator auto it = std::ranges::find_if(buttons, is_selected);

    if (it != std::end(buttons)) (*it).draw();
};
```

Compiler error if changed to something that doesn’t return a type that models forward_iterator

Iterator Concepts

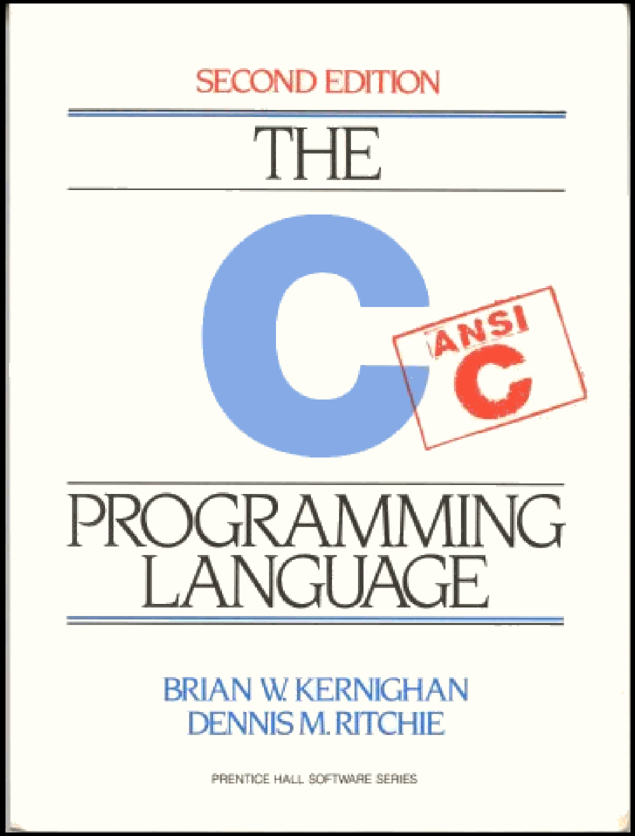
Concept	Example range
`input_iterator`	file stream opened for reading
`output_iterator`	file stream opened for writing
`forward_iterator`	singly linked list
`bidirectional_iterator`	doubly linked list
`random_access_iterator`	deque
`contiguous_iterator`	array

A Minimal C++20 Input Iterator

```
struct my_minimal_input_iterator
{
    int* p;

    using value_type = int;                          // Use std::iter_value_t<T> to retrieve
    using difference_type = std::ptrdiff_t;  // Use std::iter_difference_t<T> to retrieve

    // Non-copyable
    my_minimal_input_iterator(my_minimal_input_iterator const&) = delete;
    my_minimal_input_iterator& operator=(my_minimal_input_iterator const&) = delete;

    // Movable
    my_minimal_input_iterator(my_minimal_input_iterator&&) = default;
    my_minimal_input_iterator& operator=(my_minimal_input_iterator&&) = default;

    // Dereference (input-only, so const is ok)
    int const& operator*() const { return *p; }

    // Pre-increment
    my_minimal_input_iterator& operator++() { ++p; return *this; }

    // Post-increment (must increment but does not need to return a value)
    void operator++(int) { ++p; }
};

static_assert(std::input_iterator<my_minimal_input_iterator>);
```

Writing a Generic Algorithm

Back to K&R



SECOND EDITION
THE
C
ANSI
C
PROGRAMMING
LANGUAGE
BRIAN W. KERNIGHAN
DENNIS M. RITCHIE
PRENTICE HALL SOFTWARE SERIES

strlen, Version 1

```c
/* strlen:  return length of s */
int strlen(char[] s)
{
    int i;

    i = 0;
    while (s[i] != '\0')
        ++i;
    return i;
}
```

[The C Programming Language, §2.3 – Constants, p 39]

strlen, Version 2

```c
/* strlen:  return length of string s */
int strlen(char *s)
{
    int n;

    for (n = 0; *s != '\0'; s++)
        n++;
    return n;
}
```

[The C Programming Language, §5.3 – Pointers and Arrays, p 99]

strlen, Version 3

```c
/* strlen:  return length of string s */
int strlen(char *s)
{
    char *p = s;

    while (*p != '\0')
        p++;
    return p - s;
}
```

[The C Programming Language, §5.4 – Address Arithmetic, p 103]

The Law of Useful Return

“A procedure should return all the potentially useful information it computed.”

[From Mathematics to Generic Programming, §11.2 – Permutation Algorithms, p 202]

The Law of Useful Return

```
/* strlen:  return length of string s */
int strlen(char *s)
{
    char *p = s;

    while (*p != '\0')
        p++;
    return p - s;
}
```

Could be of use to the caller

Refactoring

```
/* find_eos:  return end of string s */
char *find_eos(char *s)
{
    char *p = s;

    while (*p != '\0')
        p++;
    return p;
}

/* strlen:  return length of string s */
int strlen(char *s)
{
    return find_eos(s) - s;
}
```

Optimization

```c
/* find_eos:  return end of string s */
char *find_eos(char *s)
{
    while (*s != '\0')
        s++;
    return s;
}
```

```c
/* strlen:  return length of string s */
int strlen(char *s)
{
    return find_eos(s) - s;
}
```

Generalization, Documentation

```c
/* find_char_unguarded:  return position of first c in string s */
/* precondition:  c exists in string s */
char *find_char_unguarded(char *s, char c)
{
    while (*s != c)
        s++;
    return s;
}


/* find_eos:  return end of string s */
char *find_eos(char *s)
{
    return find_char_unguarded(s, '\0');
}
```

Generalization, C++ Template

```
/* find_unguarded:  return position of value in array starting from first */
/* precondition:  value exists between first and end of array */
template <typename T>
T* find_unguarded(T* first, T value)
{
    while (*first != value)
        ++first;
    return first;
}
```

Analysis: Requirements of type T

```
/* find_unguarded:  return position of value in array starting from first */
/* precondition:  value exists between first and end of array */
template <typename T>
T* find_unguarded(T* first, T value)
{
    while (*first != value)
        ++first;
    return first;
}
```

Could incur a copy-construction

Inequality-comparison

Constructible (can take the address of)

Optimization: Pass by const-ref

```
/* find_unguarded:  return position of value in array starting from first */
/* precondition:  value exists between first and end of array */
template <typename T>
T* find_unguarded(T* first, T const& value)
{
    while (*first != value)
        ++first;
    return first;
}
```

Inequality-comparison

Constructible (can take the address of)

What’s Missing Here?

```
/* my_minimal_type:  should work with find_unguarded */
struct my_minimal_type
{
    // impl
};

bool operator!=(my_minimal_type const& t, my_minimal_type const& u)
{
    // impl
}
```

The Law of Completeness

“When designing an interface, consider providing all the related procedures.”

[From Mathematics to Generic Programming, §11.2 – Permutation Algorithms, p 203]

Does This Code Look Natural?

```
void modify(my_minimal_type& t, my_minimal_type& u)
{
    if (!(t != u)) return; // exit if t == u
    // impl
}
```

Completing the Interface

```
/* my_minimal_type:  should work with linear search algorithms */
struct my_minimal_type
{
    // impl
};

bool operator==(my_minimal_type const& t, my_minimal_type const& u)
{
    // impl
}

bool operator!=(my_minimal_type const& t, my_minimal_type const& u)
{
    return !(t == u);
}

static_assert(std::equality_comparable<my_minimal_type>);
```

Constraining T

```
/* find_unguarded:  return position of value in array starting from first */
/* precondition:  value exists between first and end of array */
template <std::equality_comparable T>
T* find_unguarded(T* first, T const& value)
{
    while (*first != value)
        ++first;
    return first;
}
```

T constrained by concept

Many compiler errors no longer caught here

The purpose of the constraint is not to specify the *implementation* but to to specify the *meaning* of the algorithm!

equality_comparable Requirements

Consider two values `t`, `u` of type `T`:

***Syntactically*, these expressions must be valid:**

```
bool(t == u)
bool(u == t)
bool(t != u)
bool(u != t)
```

equality_comparable Requirements

Consider two values `t`, `u` of type `T`:

<u>Semantically</u>**, the expressions reflect *equality*:**

`bool(t == u) == true` *iff operands are equal*
`bool(u == t) == true` *iff operands are equal*
`bool(t != u) == true` *iff operands are not equal*
`bool(u != t) == true` *iff operands are not equal*

equality_comparable Requirements

Consider two values `t`, u of type T:

Contractually, **the expressions must have the same** ***domain***:

```
t == u
u == t
t != u
u != t
```

All have the same preconditions.

(Domain cannot be fully unit-tested in the general case, should be documented if applicable.)

equality_comparable Requirements

Consider two values `t`, `u` of type `T`:

Further, the expressions must be *equality-preserving* and *stable*:

```
t == u
u == t
t != u
u != t
```

All can be called multiple times without modifying `t` or `u` and always returning the same values.

(This implies that the operator parameters could be const&.)

This Function Has a Bug

```
bool operator==(my_type const& t, my_type const& u)
{
    return false;
}
```

This Function Has a Bug

```
bool operator==(my_type const& t, my_type const& u)
{
    return false;
}

void my_unit_test()
{
     my_type t{};
     assert(t == t);
}
```

Does this Function Have a Bug?

```
bool operator==(my_type const& t, my_type const& u)
{
    return true;
}
```

Testing Semantic Requirements

```
// assert_equality_comparable:  unit test for std::equality_comparable values
// precondition:  t, u, v are all in the domain of == and !=
template <std::equality_comparable T>
void assert_equality_comparable(T const& t, T const& u, T const& v)
{
    assert(bool(t != u) == !bool(t == u)); // inverse

    assert(bool(t == t)); // reflexivity
    assert(!bool(t != t)); // anti-reflexivity

    assert(bool(t == u) == bool(u == t)); // symmetry
    assert(bool(t != u) == bool(u != t)); // symmetry

    if (bool(t == u) && bool(u == v)) {
        assert(bool(t == v)); // transitivity
    }
}
```

Complexity Requirements

Consider two values `t`, u of type T:

There is an implicit *complexity*, requirement:

```
t == u
u == t
t != u
u != t
```

Must be linear (worst case) in the *area* (i.e. total size of all parts) of the objects.

(Average-case complexity of equality is often nearly constant, since unequal objects tend to test unequal in an early part.)

Generalization: Support Other Kinds of Ranges

```
/* find_unguarded:  return position of value in array starting from first */
/* precondition:  value exists between first and end of array */
template <std::equality_comparable T>
T* find_unguarded(T* first, T const& value)
{
    while (*first != value)
        ++first;
    return first;
}
```

Generalization: Pointers to Iterators

```
/* find_unguarded:  return position of value in range starting from first */
/* precondition:  value exists between first and end of range */
template <typename I, std::equality_comparable T>
I find_unguarded(I first, T const& value)
{
    while (*first != value)
        ++first;
    return first;
}
```

Analysis: Requirements of Type I

```
/* find_unguarded:  return position of value in range starting from first */
/* precondition:  value exists between first and end of range */
template <typename I, std::equality_comparable T>
I find_unguarded(I first, T const& value)
{
    while (*first != value)
        ++first;
    return first;
}
```

Potentially a copy

Dereference into value compared with T (read-only)

Pre-increment, no subsequent access to earlier values

Constraining I

```
/* find_unguarded:  return position of value in range starting from first */
/* precondition:  value exists between first and end of range */
template <std::input_iterator I, std::equality_comparable T>
I find_unguarded(I first, T const& value)
{
    while (*first != value)
        ++first;
    return first;
}
```

Strengthening the Constraints

```
/* find_unguarded:  return position of value in range starting from first */
/* precondition:  value exists between first and end of range */
template <std::input_iterator I, std::equality_comparable T>
I find_unguarded(I first, T const& value)
{
    while (*first != value)
        ++first;
    return first;
}
```

These types are related!

Here is where they interact

The Law of Separating Types

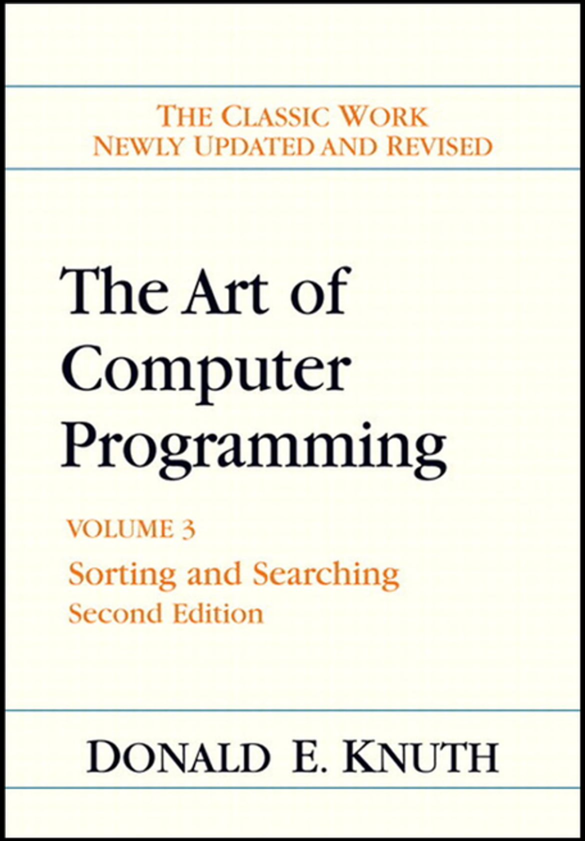
“Do not assume that two types are the same when they may be different.”

[From Mathematics to Generic Programming, §11.2 – Permutation Algorithms, p 202]

The Law of Separating Types

```
/* find_unguarded:  return position of value in range starting from first */
/* precondition:  value exists between first and end of range */
template <std::input_iterator I, std::equality_comparable_with<std::iter_value_t<I>> T>
I find_unguarded(I first, T const& value)
{
    while (*first != value)
        ++first;
    return first;
}
```

Performance vs. Safety

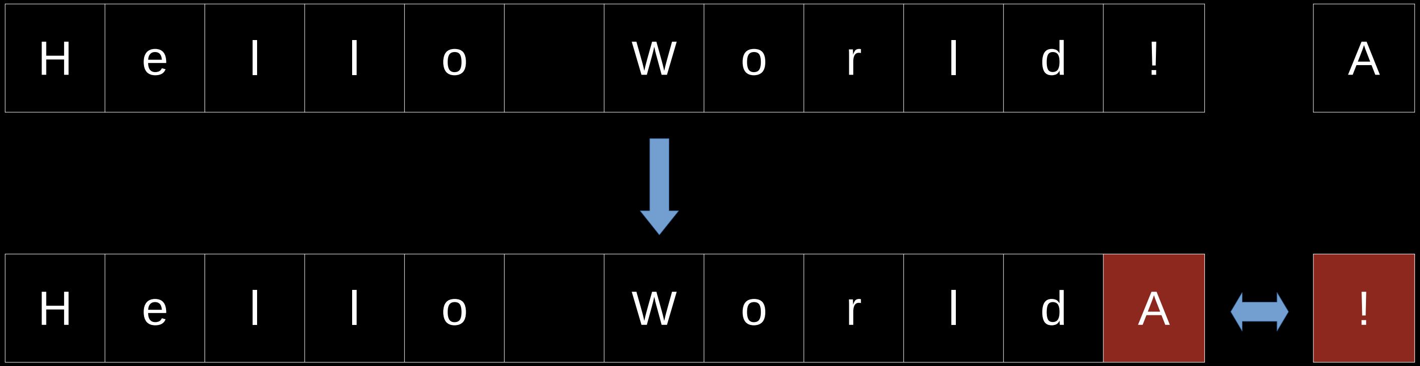


A Faster Safe find

H	e	l	l	o		W	o	r	l	d	!		A

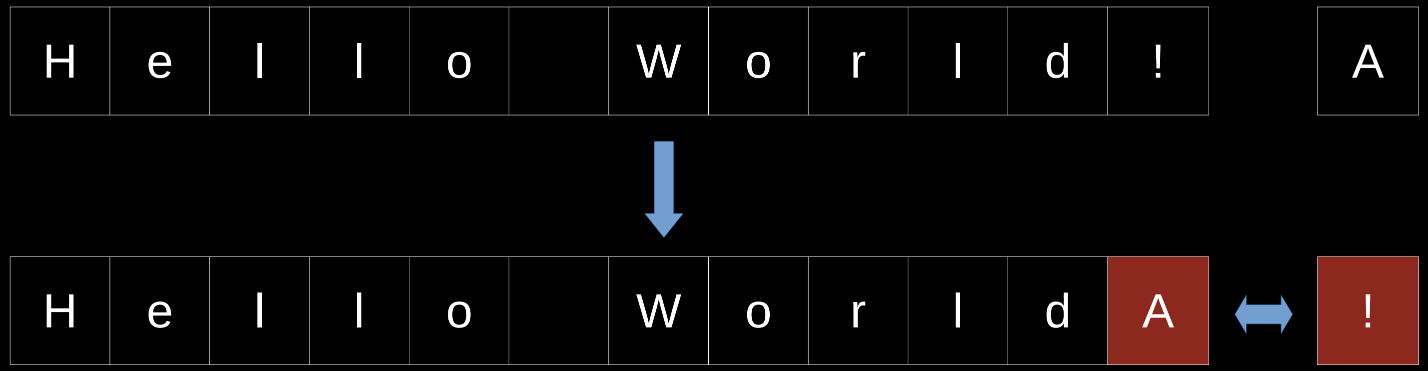
[The Art of Computer Programming, vol. 3, 2nd ed, §6.1 – Sequential Searching, pp 397-398]

A Faster Safe find



[The Art of Computer Programming, vol. 3, 2nd ed, §6.1 – Sequential Searching, pp 397-398]

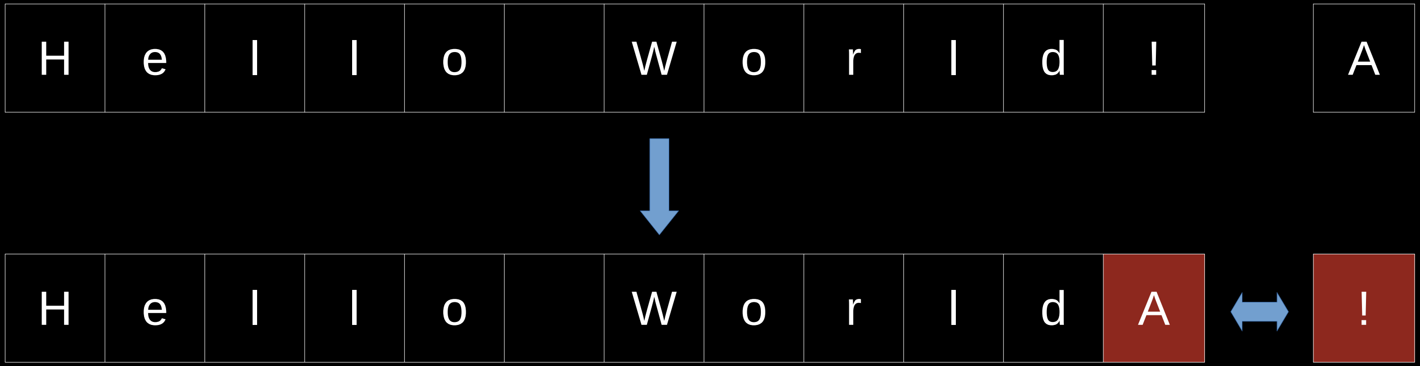
A Faster Safe find




```
find_unguarded(s, 'A');
```

[The Art of Computer Programming, vol. 3, 2nd ed, §6.1 – Sequential Searching, pp 397-398]

A Faster Safe find



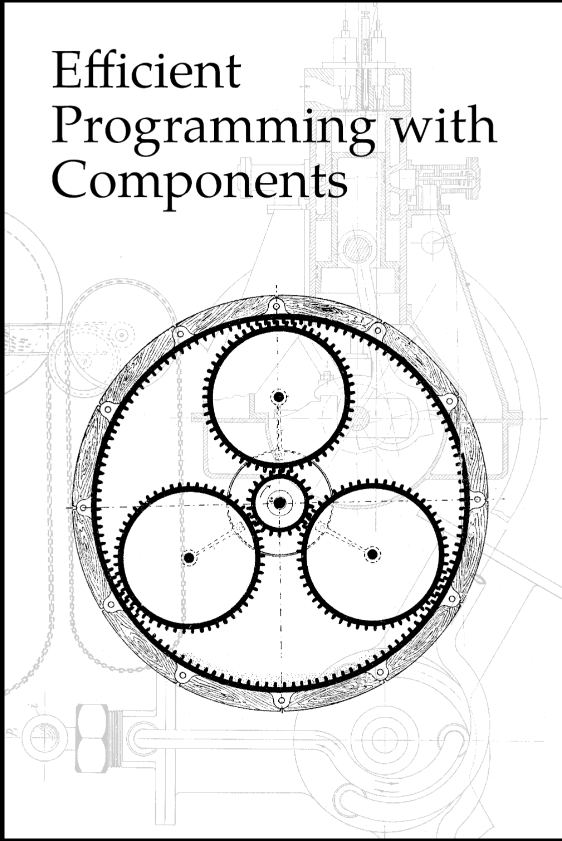

```
find_unguarded(s, 'A');
```

[The Art of Computer Programming, vol. 3, 2nd ed, §6.1 – Sequential Searching, pp 397-398]

A Faster find for Mutable Bidirectional Ranges

```
/* find_guarded:  return position of value in the range [first, last)
                  returns last if value is not found
*/
template <std::bidirectional_iterator I>
requires std::indirectly_swappable<I, std::iter_value_t<I>*> &&
    std::equality_comparable<std::iter_value_t<I>>
I find_guarded(I first, I last, std::iter_value_t<I> value)
{
    if (first == last) return first;
    --last;
    std::ranges::iter_swap(last, &value);
    first = find_unguarded(first, *last);
    std::ranges::iter_swap(last, &value);
    if (first == last && *first != value) ++first;
    return first;
}
```

Recommended Reading



https://www.jmeiners.com/efficient-programming-with-components/