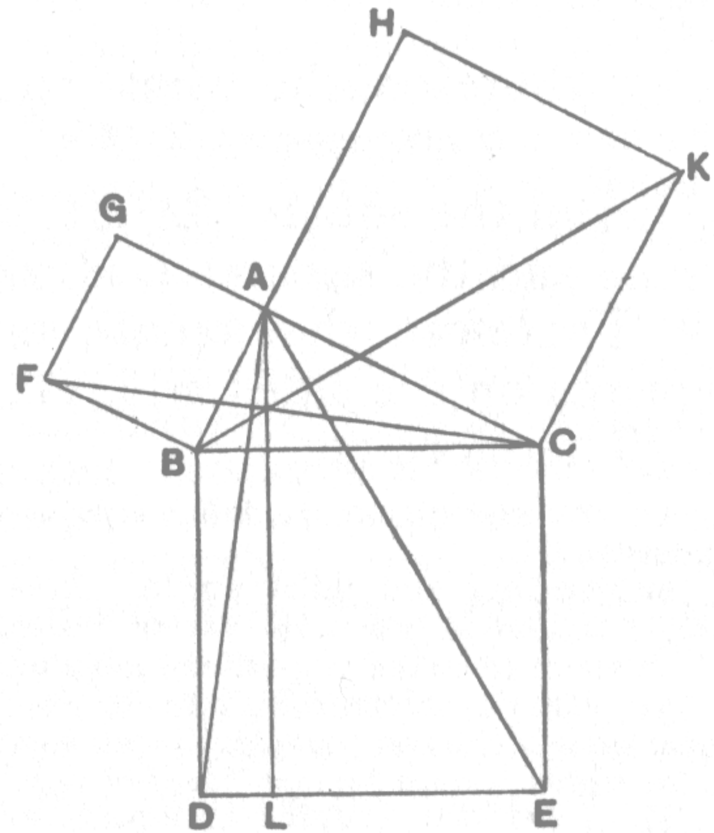
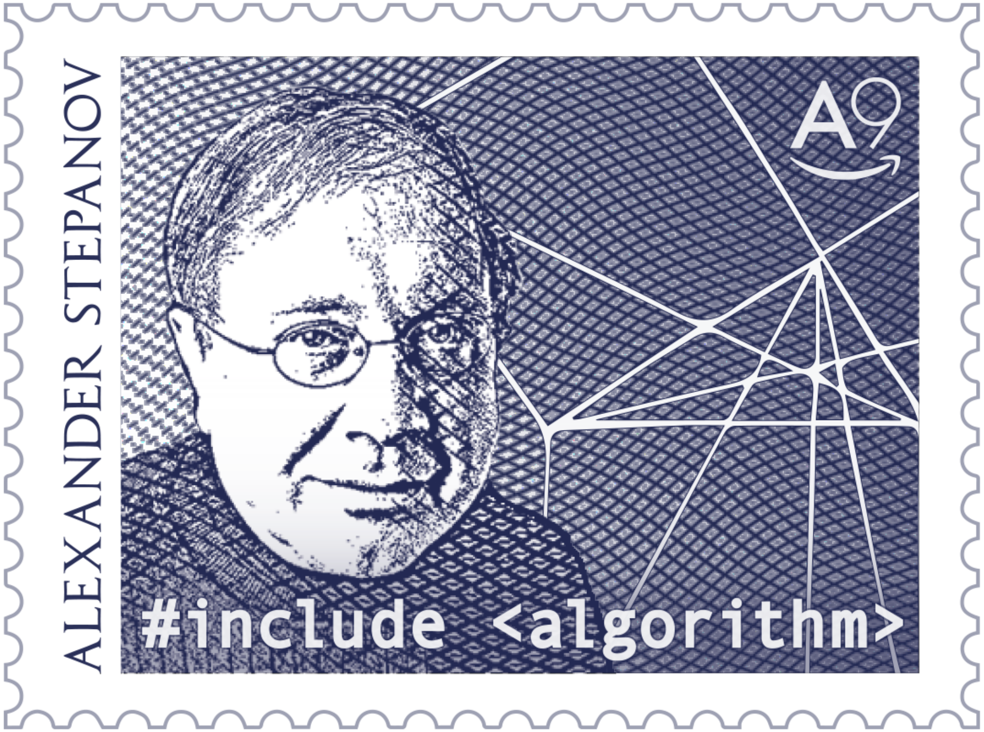
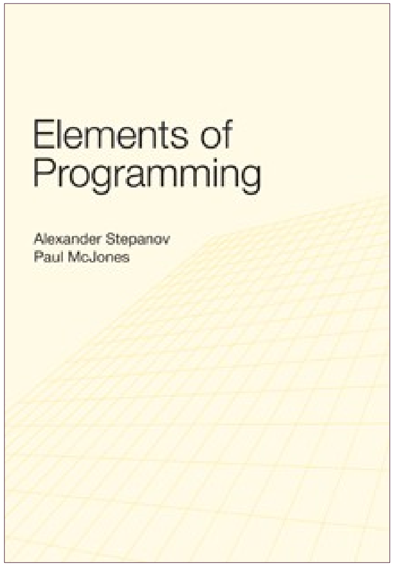
From type to concept

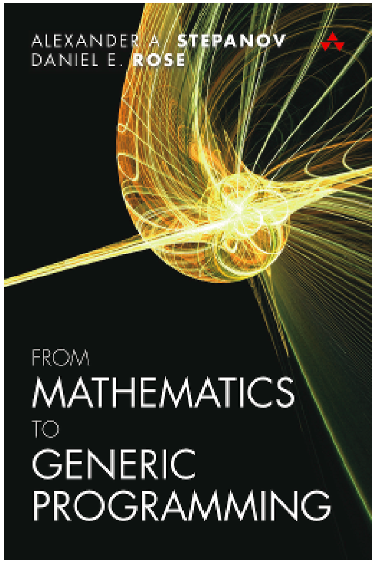
Petter Holmberg – C++ Stockholm 0x06 – September 2017



Origins


ALEXANDER STEPANOV
A9
#include <algorithm>


Elements of Programming
Alexander Stepanov
Paul McJones


ALEXANDER A. STEPANOV
DANIEL E. ROSE
FROM
MATHEMATICS
TO
GENERIC
PROGRAMMING

Concepts

Natural science	Mathematics	Programming
genus	theory	concept
species	model	type or class
individual	element	instance

From Mathematics to Generic Programming §10.3

Concepts

Integral: `int`, `uint8_t`, `int32_t`, …

Character: `uint8_t`, `char`, `wchar_t`, …

Mutex: `mutex`, `recursive_mutex`, `timed_mutex`, …

Clock: `chrono::system_clock`, `chrono::high_resolution_clock`, …

Concepts

“A ***concept*** can be viewed as a set of requirements on types, or as a predicate that tests whether types meet those requirements. The requirements concern

- The *operations* the types must provide
- Their *semantics*
- Their *time/space complexity*

A type is said to satisfy a concept if it meets these requirements.”

From Mathematics to Generic Programming §10.3

STL concepts

Container: vector<double>, map<uint64_t, string>, …

Iterator: list<bool>::iterator, istream_iterator<int>, int*, …

Invocable: min, [](char&){}, void (fn*)(), …

Regular: int, string, vector<pair<int, string>>, …

Foundations

Semiregular: `int`, `string`, `class{char, vector<double>}`, …

- Types with ***value semantics***
- Are easily understood by both programmers and compilers
- Can be passed to and returned from functions
- Compose naturally in standard data structures
- Satisfy basic requirements for standard algorithms

C++20 Concepts

```
template <typename T>
concept Semiregular = true;
```

C++20 Concepts

```
template <typename T>
concept Semiregular = true;


template <typename T>
void fn(T t) {}


template <typename T>
struct s {
    T t;
};


template <typename T>
constexpr T v = {1, 2, 3};
```

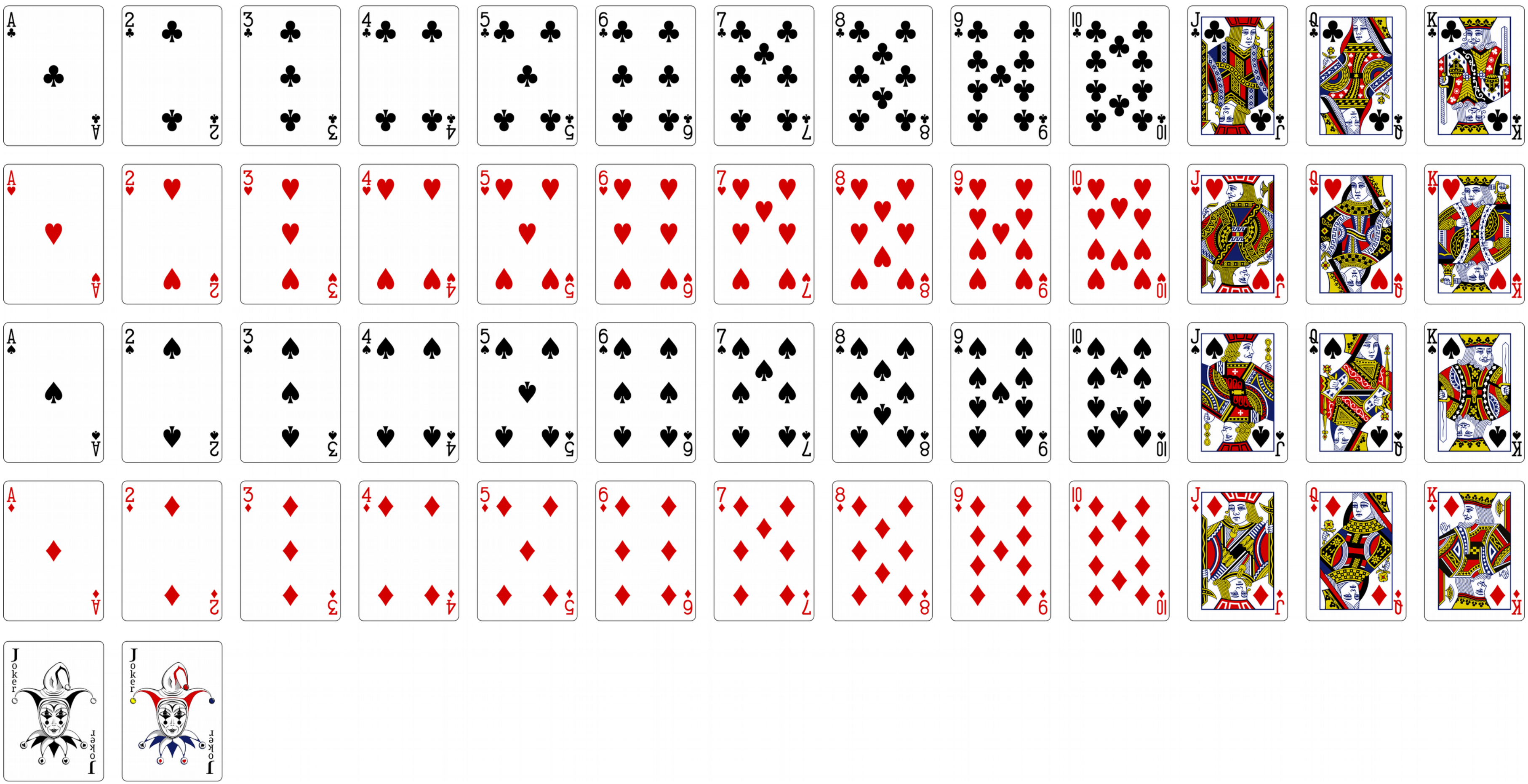
C++20 Concepts

```
template <typename T>
concept Semiregular = true;


template <typename T>
    requires Semiregular<T>
void fn(T t) {}

template <typename T>
    requires Semiregular<T>
struct s {
    T t;
};

template <typename T>
    requires Semiregular<T>
constexpr T v = {1, 2, 3};
```

C++ Extensions for Concepts TS

```
template <typename T>
concept Semiregular = true;


// C++ Extensions for Concepts TS fallback (GCC6+)
#define concept    concept bool


// Without concepts
template <typename T>
    // requires Semiregular<T>
void fn(T t) {}
```

An example type



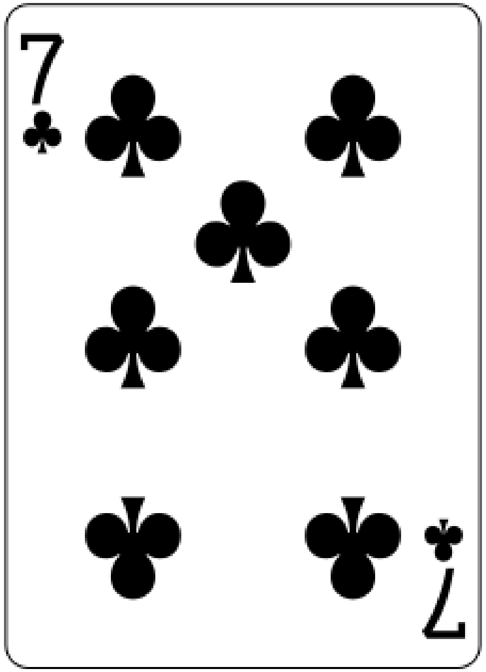
Vectorized Playing Cards 2.0 - http://sourceforge.net/projects/vector-cards/

An example type

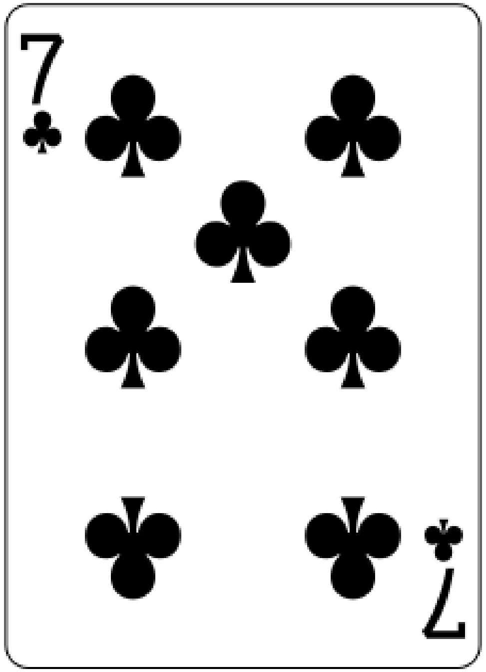
```
enum class rank_t {
    joker = 0,
    ace_low = 1,
    // pip cards not named
    jack = 11,
    queen = 12,
    king = 13,
    ace_high = 14
};

enum class suit_t {
    none,
    clubs,
    diamonds,
    hearts,
    spades
};
```

```
struct card {
    rank_t rank;
    suit_t suit;
};
```

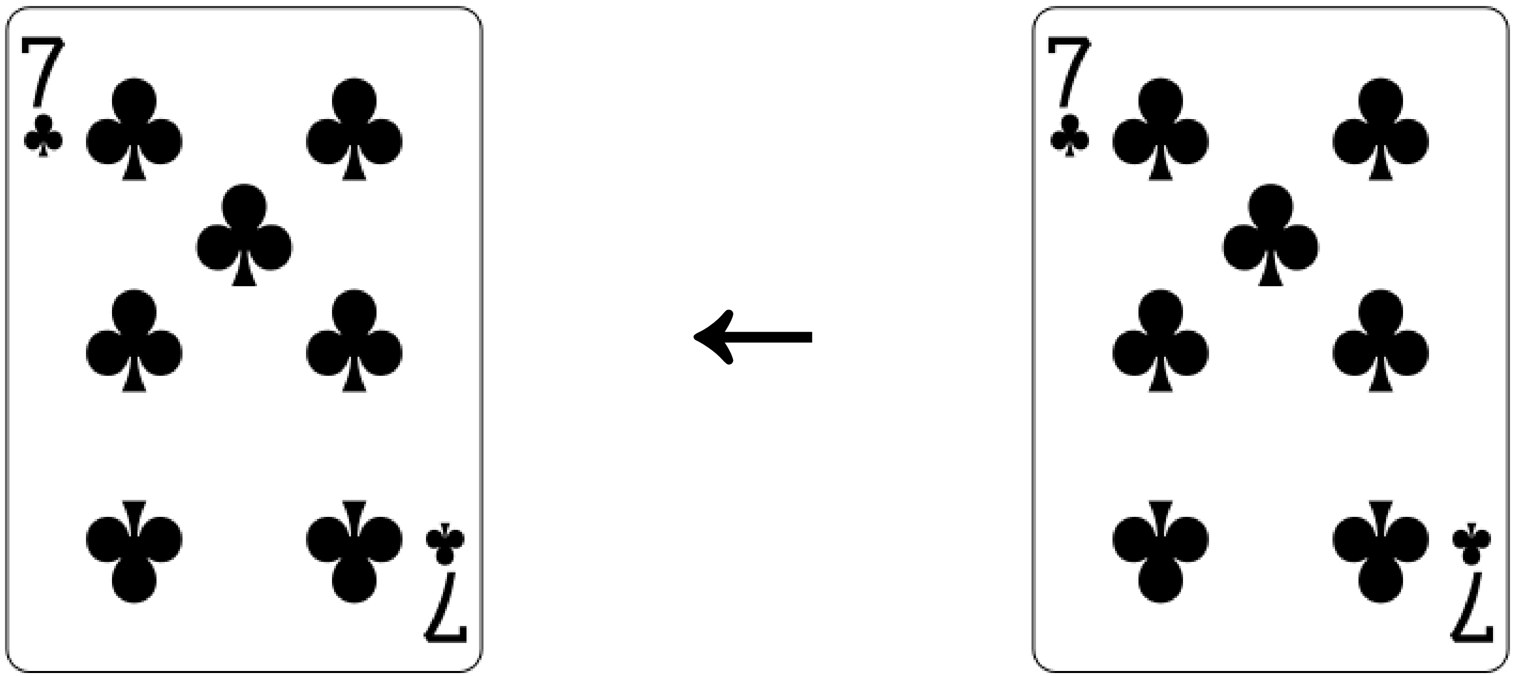

Q
Q


7
7

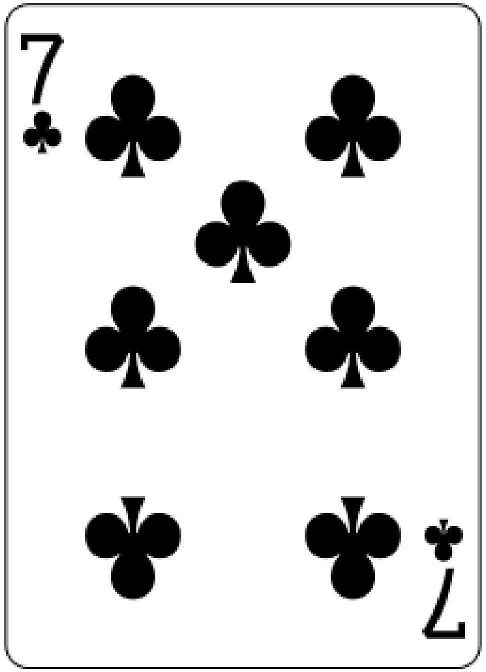
Copy assignment



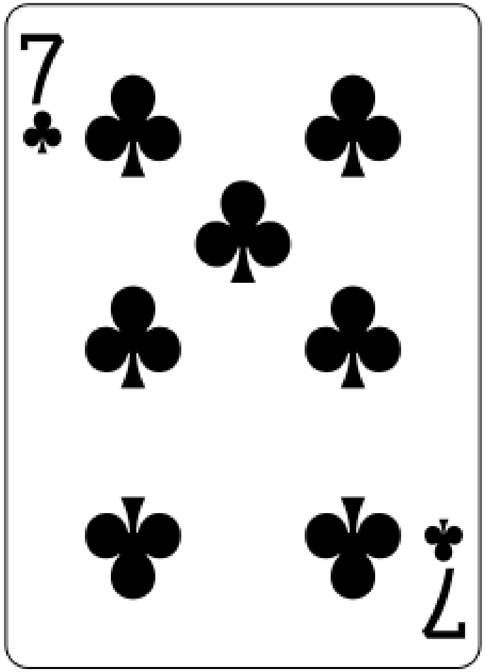
Copy assignment



Copy assignment



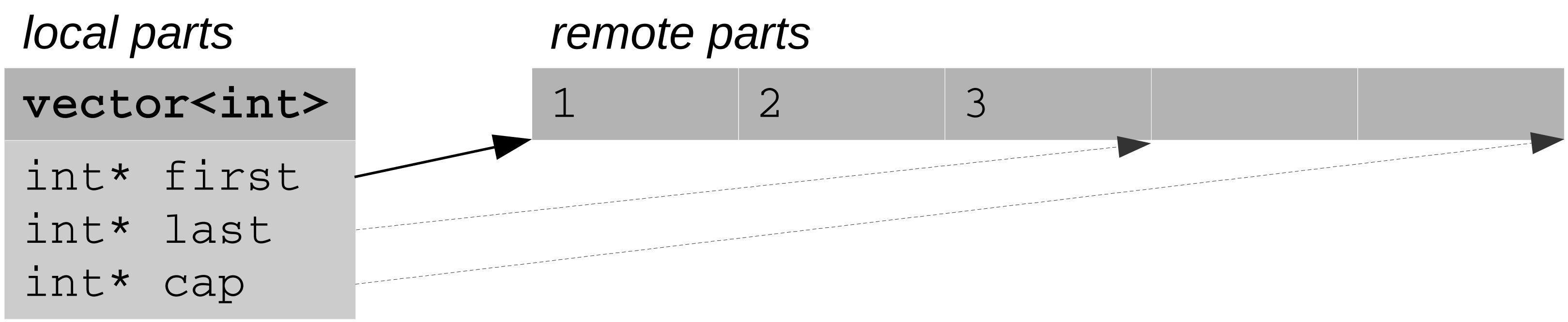
=



```
struct card {
    rank_t rank;
    suit_t suit;

    card& operator=(const card& a) {
        rank = a.rank;
        suit = a.suit;
        return *this;
    }
};
```

```
template <typename T>
concept Assignable =
    requires (T a, const T& b) {
        { a = b } -> T&;
    };
    // axiom copy_semantics:
    //     eq(a = b, b)
    // complexity:
    //     O(areaof(b))
```

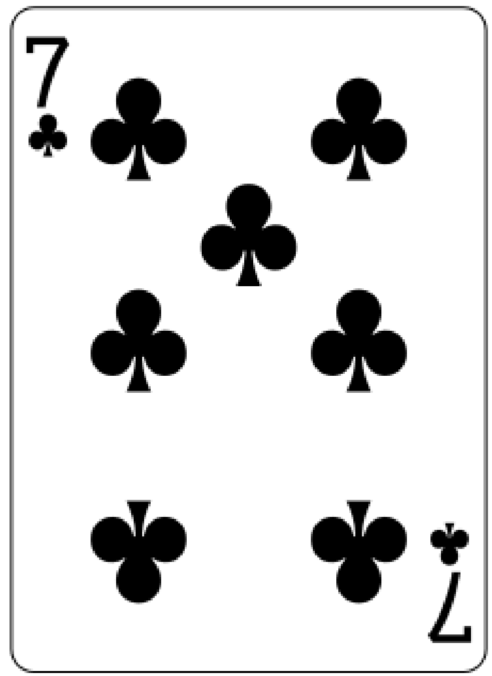
Composite objects



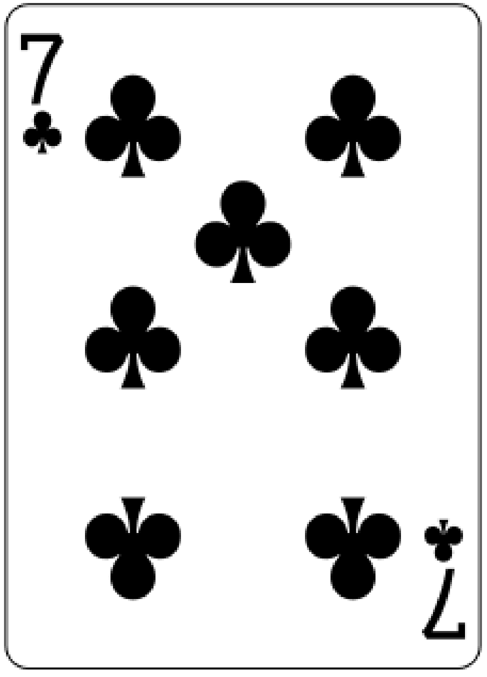

```
vector<int> v {1, 2, 3};

assert(sizeof(v) == 3 * sizeof(int*));
assert(areaof(v) == sizeof(v) + 3 * sizeof(int));
```

Elements of Programming §12.1

```
template <typename T>
concept Semiregular =
    Assignable<T>;
```


7

Destruction

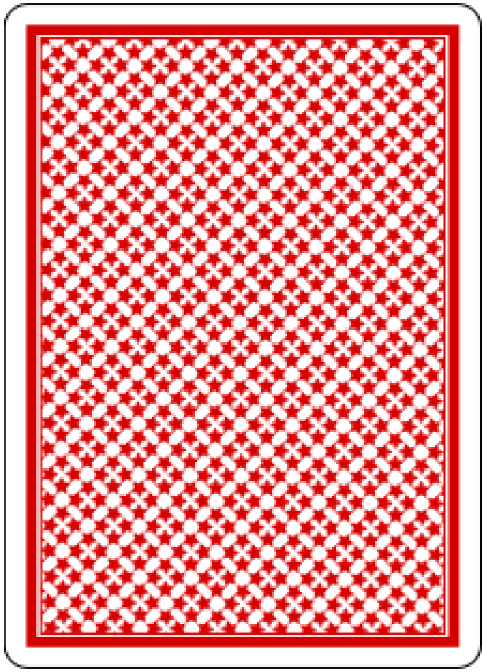


Destruction

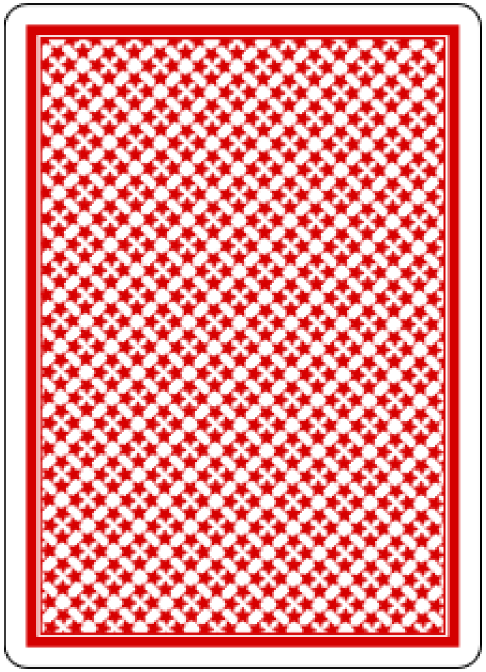
```
struct card {
    rank_t rank;
    suit_t suit;
    card& operator=(const card&);

    ~card() {}
};
```

```
template <typename T>
concept Destructible =
    std::is_nothrow_destructible_v<T>;
    // axiom end_of_object_lifetimes:
    //     see §3.4, §12.4 in ISO/IEC N4296
    // complexity:
    //     O(areaof(a))
```

```
template <typename T>
concept Semiregular =
    Assignable<T> &&
    Destructible<T>;
```



Default construction



Default construction

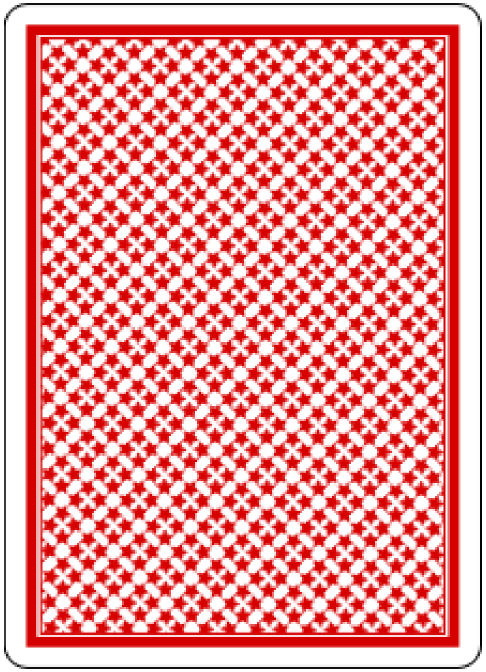
```
void fn() {
    int i;      // what is the value of i?
    int j = i; // undefined behavior
    i = 52;     // OK to assign to i
}               // OK to destruct i
```

Default construction

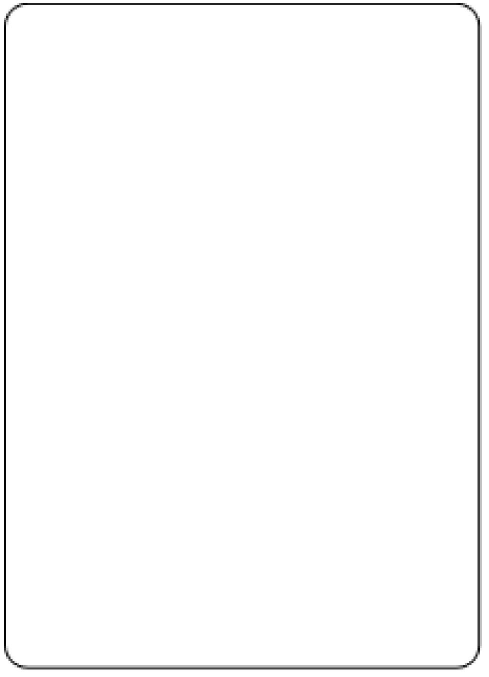
“An object is in a ***partially formed*** state if it can be assigned to or destroyed. For an object that is partially formed but not ***well formed***, the effect of any procedure other than assignment (only on the left side) and destruction is not defined.”

Elements of Programming §1.5

Default construction



Default construction



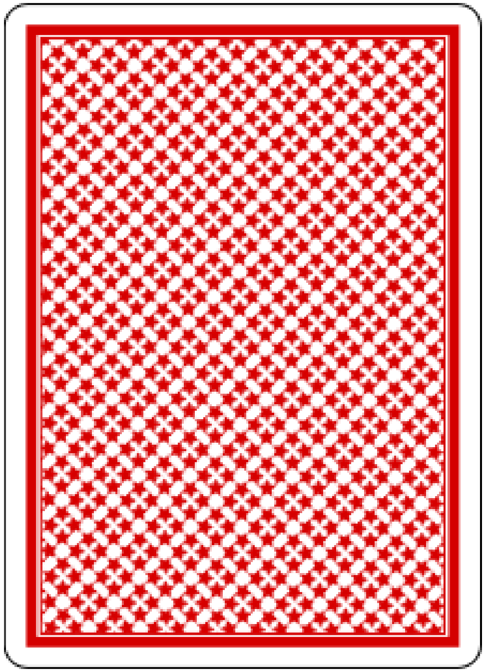
```
struct card {
    rank_t rank;
    suit_t suit;

    card& operator=(const card& a) {
        rank = a.rank;
        suit = a.suit;
        return *this;
    }

    ~card() {}

    card() {}
};
```

```
template <typename T, typename ...Args>
concept Constructible =
    Destructible<T> &&
    std::is_constructible_v<T, Args...>;
```

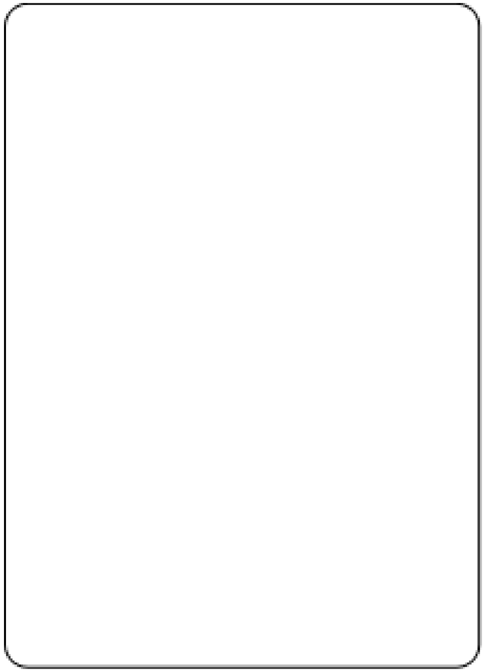
```
template <typename T>
concept Default_constructible =
    Constructible<T>;
```

```
template <typename T>
concept Semiregular =
    Assignable<T> &&
    Default_constructible<T>;
    // axiom partially_formed:
    //      T a is not necessarily well-formed
    //      T a => eq(a = b, b)
    //      T a => ~a()
    // complexity of default constructor:
    //      O(sizeof(a))
```

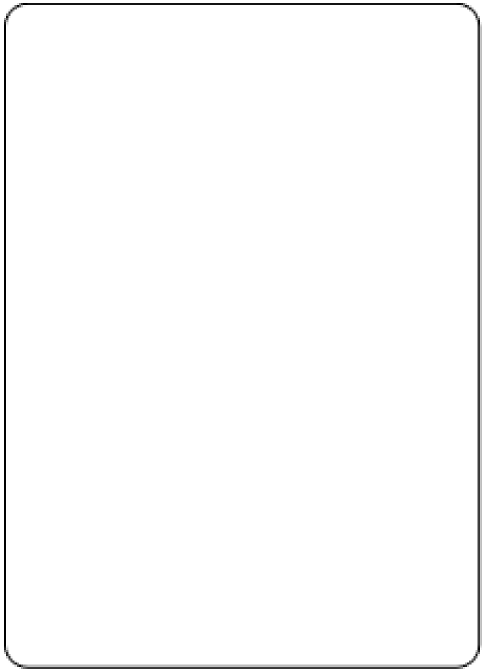
Default construction + assignment

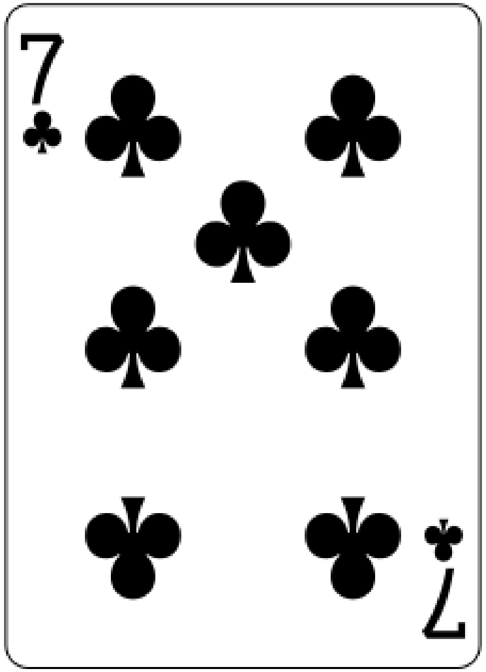


Default construction + assignment

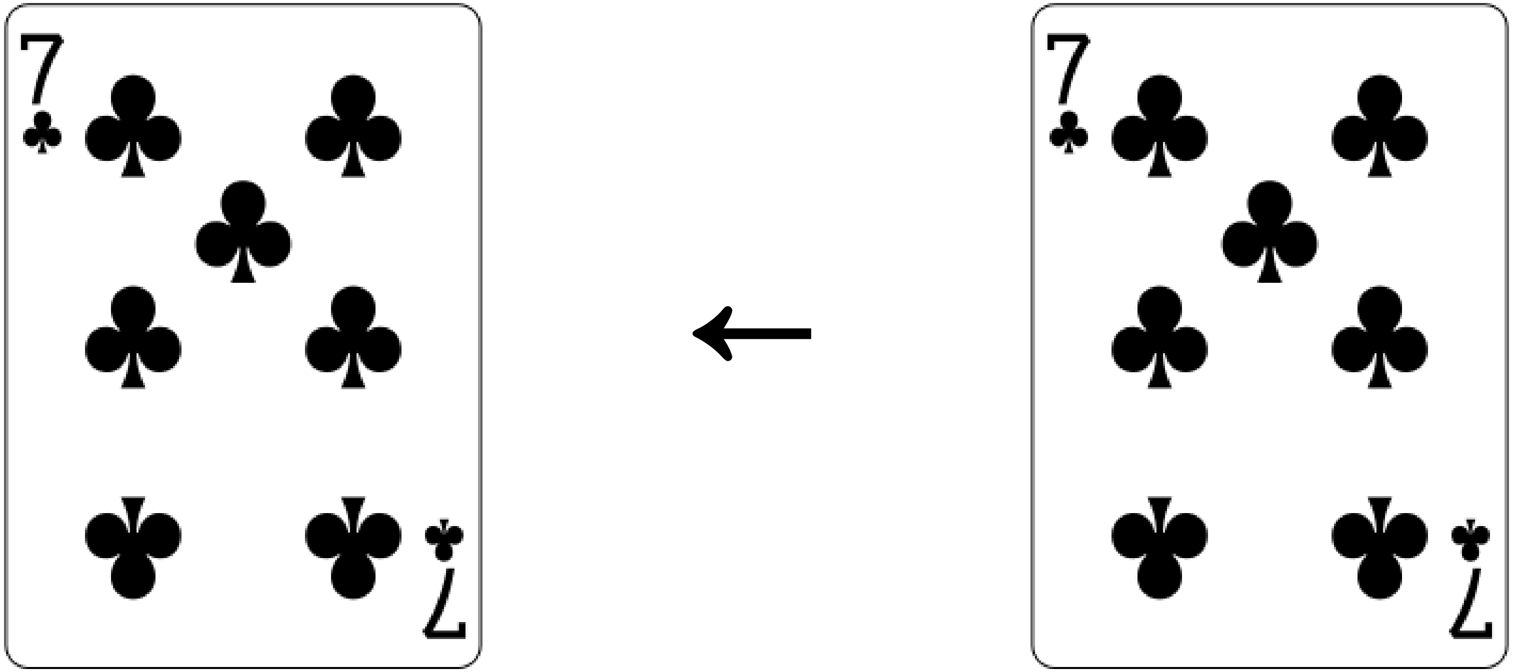


Default construction + assignment

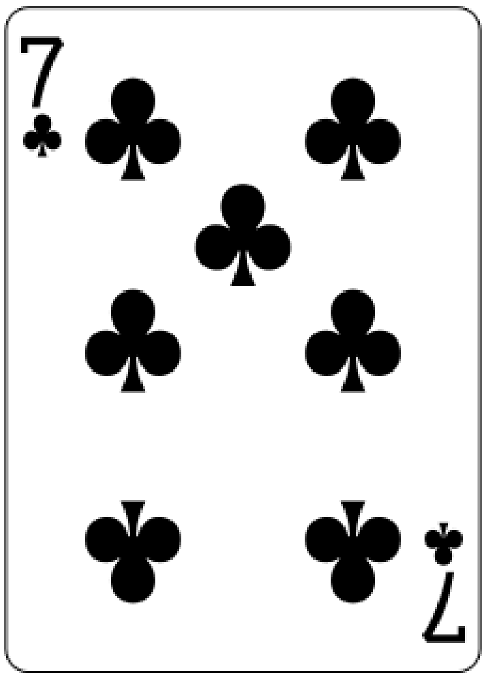




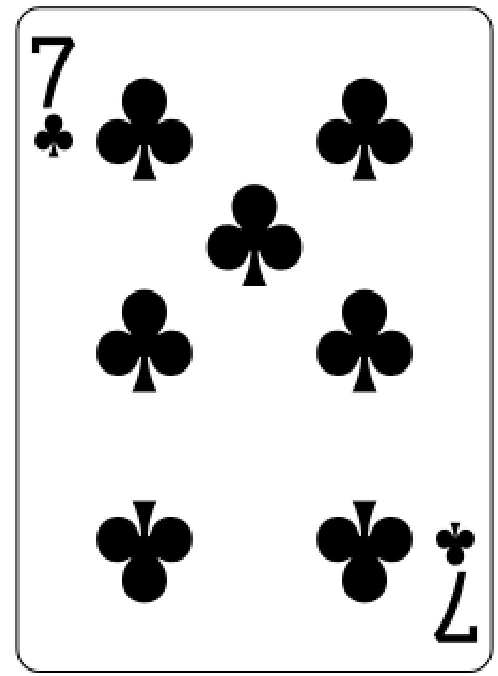
Default construction + assignment



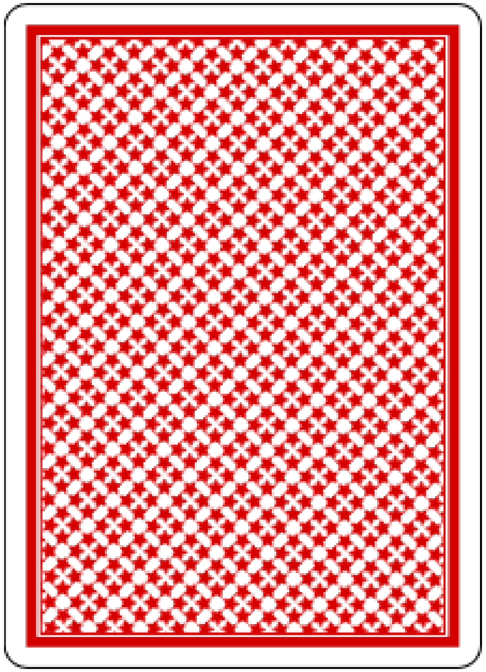
Default construction + assignment



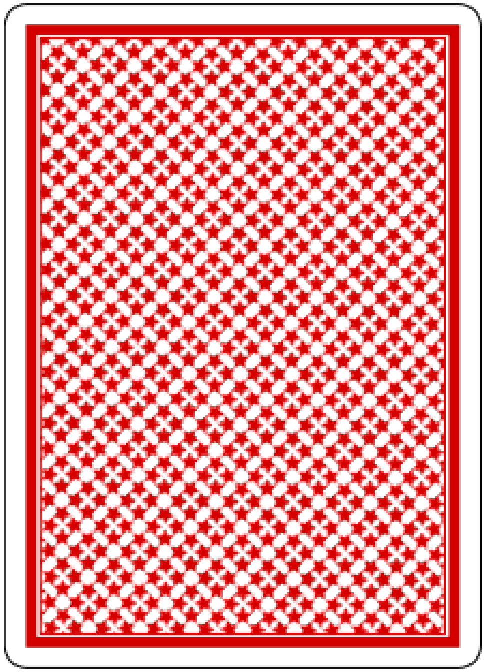
=

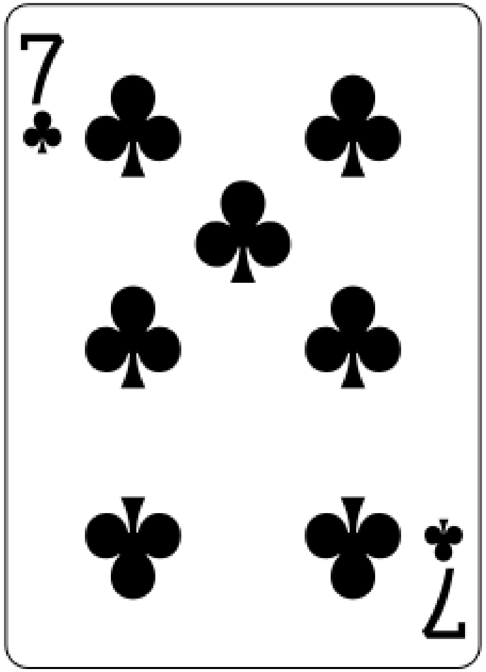


Copy construction

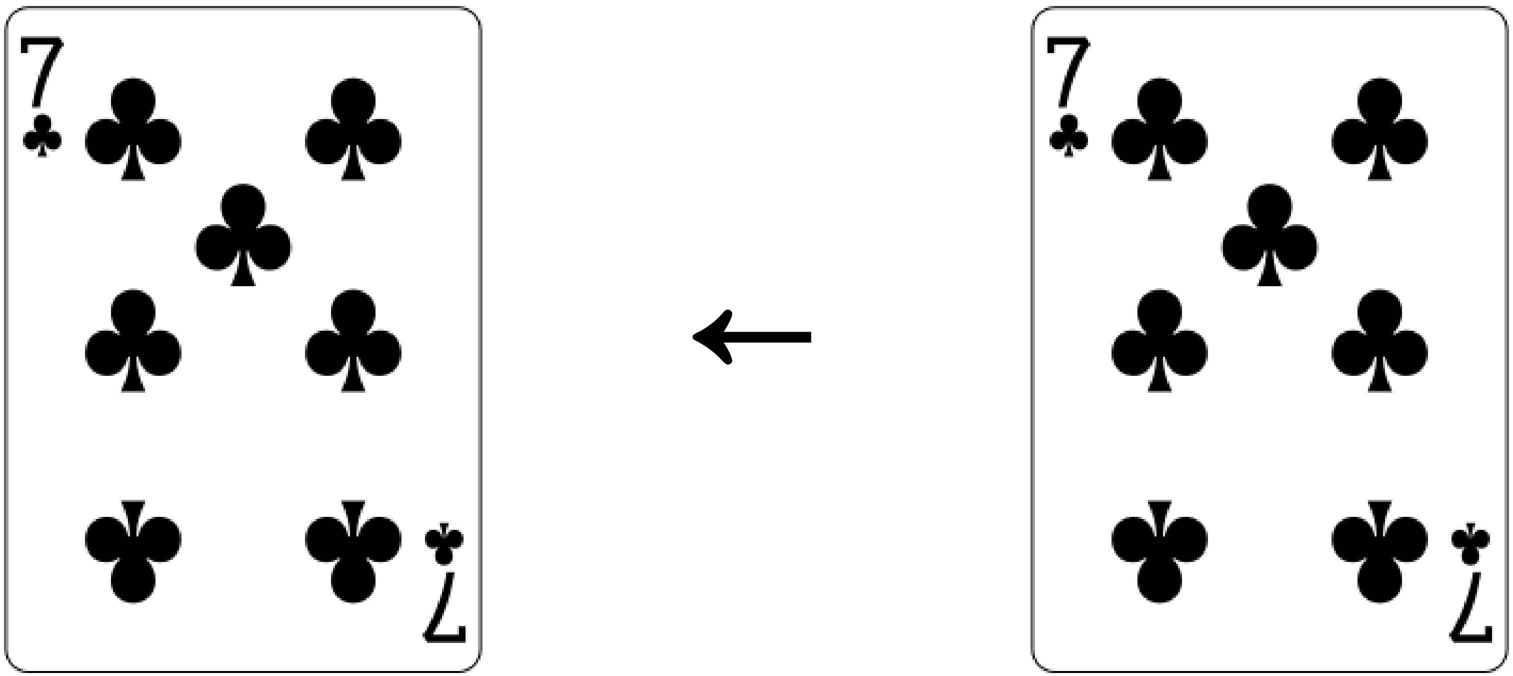


Copy construction

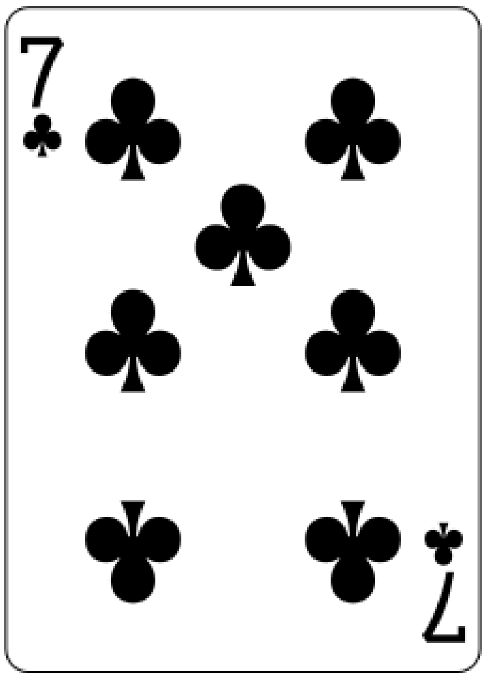




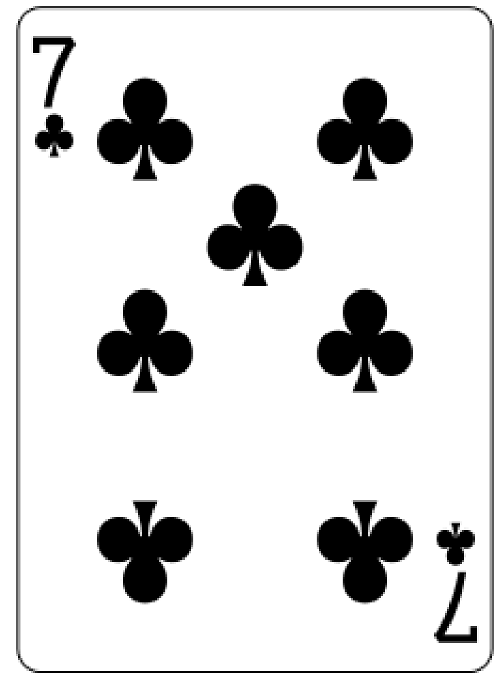
Copy construction



Copy construction



=



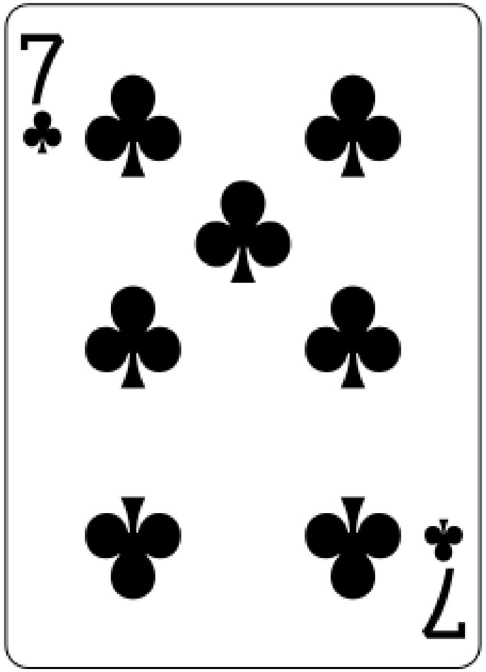
```
struct card {
    rank_t rank;
    suit_t suit;
    card& operator=(const card&);
    ~card();
    card();

    card(const card& a)
        : rank{a.rank}
        , suit{a.suit}
    {}
};
```

```
template <typename T>
concept Copy_constructible =
    Constructible<T, const T&>;
    // axiom copy_semantics:
    //     eq(T{a}, a)
    // complexity:
    //     O(areaof(a))
```

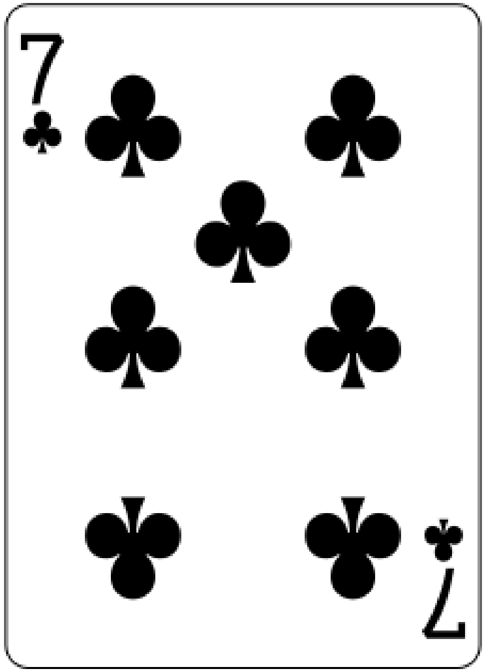
```
template <typename T>
concept Copyable =
    Copy_constructible<T> &&
    Assignable<T>;
```

```
template <typename T>
concept Semiregular =
    Copyable<T> &&
    Default_constructible<T>;
    // axiom partially_formed:
    //       T a is not necessarily well-formed
    //       T a => eq(a = b, b)
    //       T a => ~a()
    // complexity of default constructor:
    //       O(sizeof(a))
```

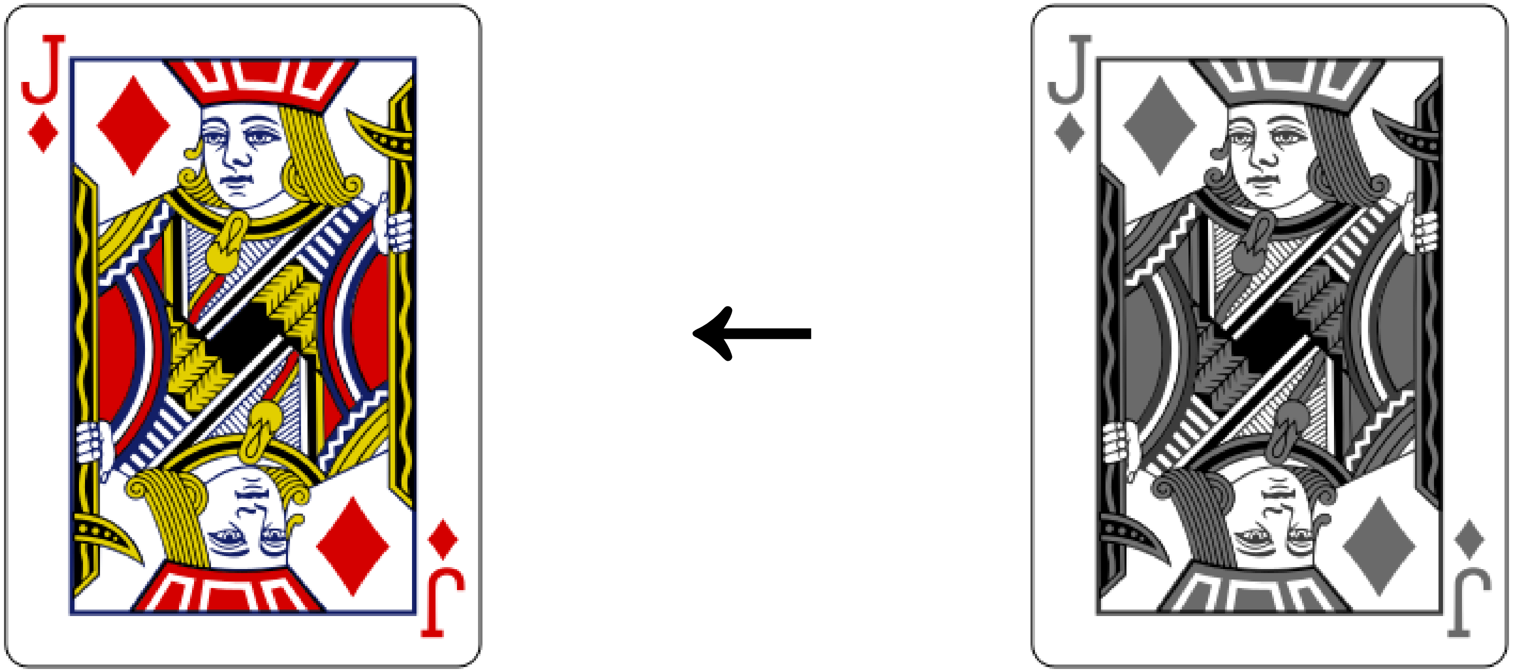
Move assignment



Move assignment



Move assignment



Standard library requirements

`MoveAssignable` requirements:

Expression	Return type	Return value	Post-condition
`T = rv`	`T&`	`t`	`t` is equivalent to the value of `rv` before the assignment
`rv`'s state is unspecified [*Note:* `rv` must still meet the requirements of the library component that is using it. The operations listed in those requirements must work as specified whether `rv` has been moved from or not. — *end note*]			

ISO/IEC N4296 §17.6.3.1, Table 22

Standard library requirements

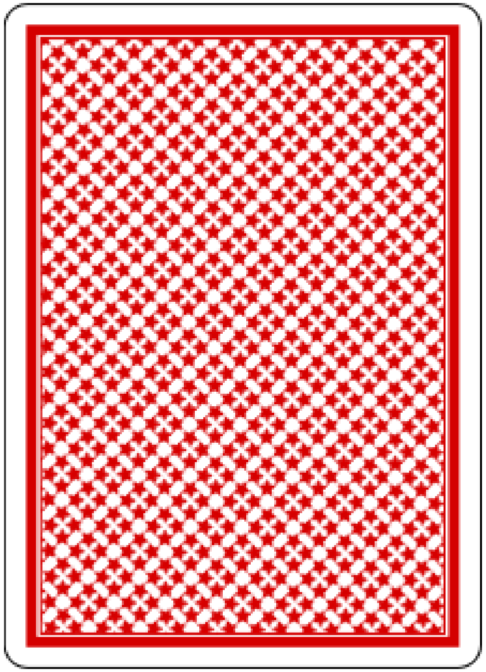
←

```
struct card {
    rank_t rank;
    suit_t suit;
    card& operator=(const card&);
    ~card();
    card();
    card(const card&);

    card& operator=(card&& a) {
        rank = a.rank;
        suit = a.suit;
        return *this;
    }
};
```

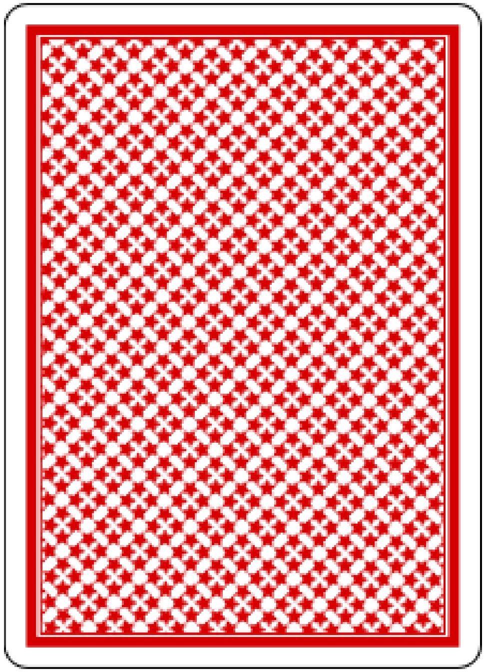
```
template <typename T, typename U>
concept Assignable =
    requires (T a, U&& b) {
        { a = std::forward<U>(b) } -> T&;
    };
    // axiom copy_semantics:
    //     eq(a = b, b)
    // axiom move_semantics:
    //     eq(a, b) => eq(a, c = std::move(b))
    //     op(a) requires no specified value =>
    //     b = std::move(a) => op(a)
    // }
    // complexity of copy assignment:
    //     O(areaof(b))
    // complexity of move assignment:
    //     O(sizeof(b))
```

```
template <typename T>
concept Semiregular =
    Copyable<T> &&
    Default_constructible<T>;
    // axiom partially_formed:
    //      T a is not necessarily well-formed
    //      T a => a = std::move(b)
    //      T a => eq(a = b, b)
    //      T a => ~a()
    // complexity of default constructor:
    //      O(sizeof(a))
```

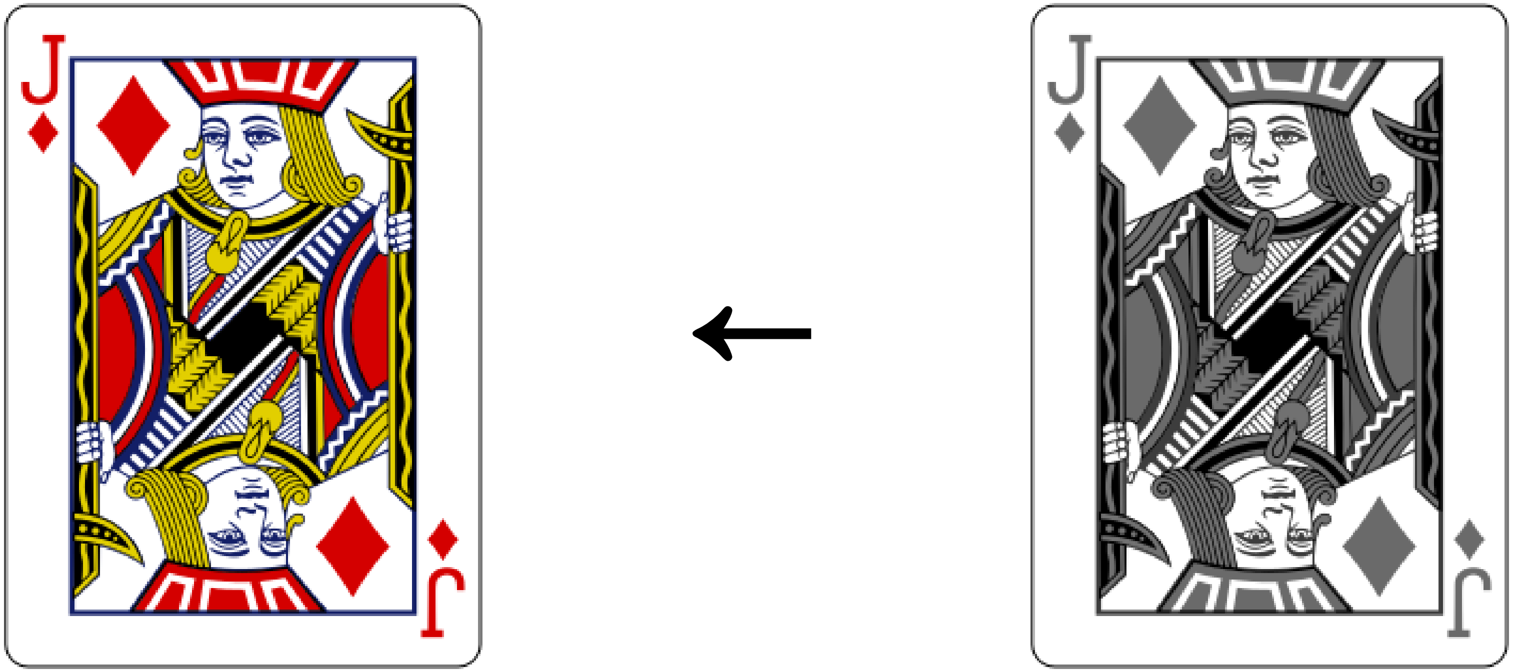
Move construction



Move construction



Move construction



```
struct card {
    rank_t rank;
    suit_t suit;
    card& operator=(const card&);
    ~card();
    card();
    card(const card&);
    card& operator=(card&&);

    card(card&& a)
        : rank{a.rank}
        , suit{a.suit}
    {}
};
```

```
template <typename T>
concept Move_constructible =
    Constructible<T, T&&>;
    // axiom move_semantics:
    //     eq(a, b) => eq(T{std::move(a)}, b)
    //     op(a) requires no specified value =>
    //     T b{std::move(a)} => op(a)
    // }
    // complexity:
    //     O(sizeof(a))
```

```
template <typename T>
concept Copy_constructible =
    Move_constructible<T> &&
    Constructible<T, const T&>;
```

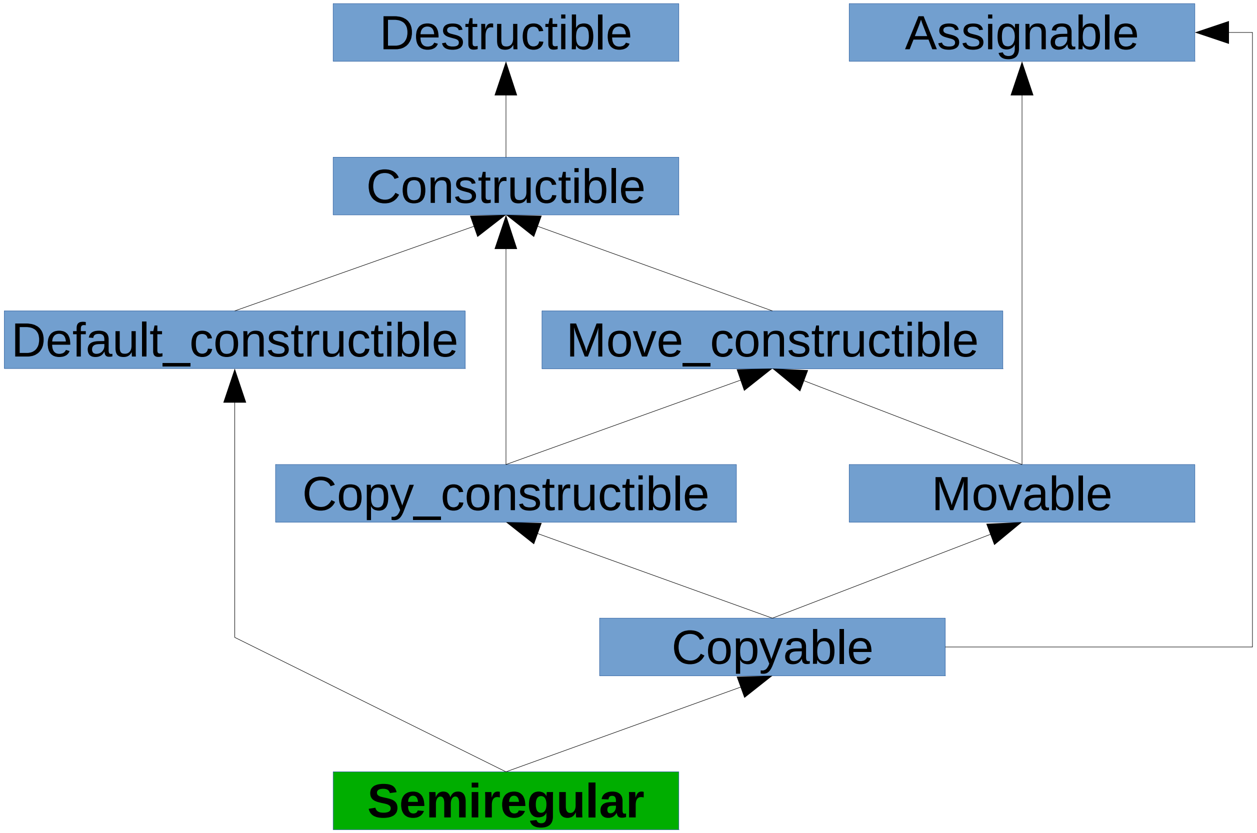
```
template <typename T>
concept Movable =
    Move_constructible<T> &&
    Assignable<T, T&&>;
```

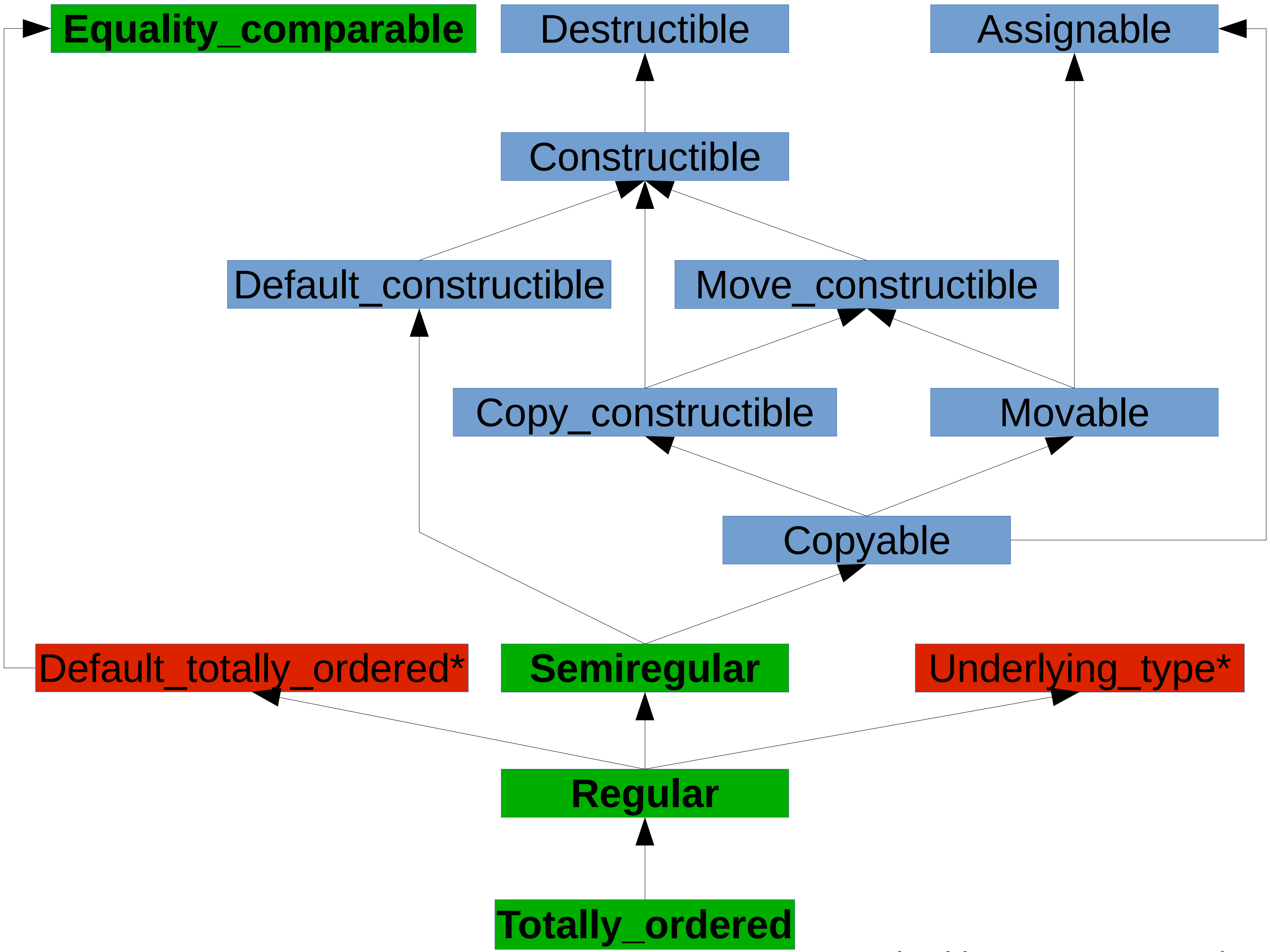
```
template <typename T>
concept Copyable =
    Movable<T> &&
    Copy_constructible<T> &&
    Assignable<T, const T&>;
```

```
template <typename T>
concept Semiregular =
    Copyable<T> &&
    Default_constructible<T>;
    // axiom partially_formed:
    //       T a is not necessarily well-formed
    //       T a => a = std::move(b)
    //       T a => a = b
    //       T a => ~a
    // complexity of default constructor:
    //       O(sizeof(a))
```

```
struct card {
    rank_t rank;
    suit_t suit;
    card& operator=(const card& a) {
        rank = a.rank; suit = a.suit; return *this;
    }
    ~card() {}
    card(const card& a) : rank{a.rank}, suit{a.suit} {}
    card() {}
    card(card&& a) : rank{a.rank}, suit{a.suit} {}
    card& operator=(card&& a) {
        rank = a.rank; suit = a.suit; return *this;
    }
};
```

```
struct card {
    rank_t rank;
    suit_t suit;
};
```


Destructible
Assignable
Constructible
Default_constructible
Move_constructible
Copy_constructible
Movable
Copyable
Semiregular



*Required by Stepanov, not in std proposals

“One of your central goals as a programmer should be to identify existing concepts in your application. You will often develop new algorithms, occasionally develop a new data structure, and only rarely define a new concept. In that rare situation, a lot of work is needed to ensure that it is a true concept and not just a collection of unrelated requirements.”

From Mathematics to Generic Programming §10.9

More information

Generic programming
Alexander A. Stepanov, Paul McJones, Dave Musser et al.
http://stepanovpapers.com/

Overload Journal article series on concepts, issues #129, #131, #136
Andrew Sutton
https://accu.org/index.php/journals/c78/

Concepts: The Future of Generic Programming
Bjarne Stroustrup
http://stroustrup.com/good_concepts.pdf

STL2 (“C++ Extensions for Ranges")
Casey Carter, Eric Niebler
https://github.com/CaseyCarter/cmcstl2