Structs

```
struct strange {
    int n;
    bool b;
    std::vector<int> a = std::vector<int> (0);
};
int main () {
    strange x = {1, true, {1,2,3}};
    strange y = x; // all elements are copied
    std::cout << y.n << "  " << y.a[2] << "\n";
    // outputs: 1  3
    return 0;
}
```

member var.

Geometry Exercise

```
 3 // Datastructure representing a 3D vector
 4 struct vec {
 5   double x;
 6   double y;
 7   double z;
 8 };
 9 void print_vec(const vec& v) {
10   std::cout << "(" << v.x << "," << v.y << "," << v.z << ")";
11 }
12 // POST: returns the sum of a and b
13 vec sum(const vec& a, const vec& b) {
14   return {a.x + b.x, a.y + b.y, a.z + b.z};
15   // vec tmp;
16   // tmp.x = a.x + b.x;
17   // tmp.y = a.y + b.y;
18   // tmp.z = a.z + b.z;
19   // return tmp;
20 }
21 // Subtask 2: defining a line in 3D--------------------
22 struct line {
23   vec start;
24   vec end;  // INV: start != end
25 };
26 // helper function to print a vector
27 void print_line(const line& l) {
28   print_vec(l.start);
29   std::cout << " <-> ";
30   print_vec(l.end);
31 }
```

Geometry Exercise

```
36  // POST: returns a new line obtained by shifting l by v.
37  line shift_line(const line& l, const vec& v) {
38    return {sum(l.start, v), sum(l.end, v)};
39  }
40  // Subtask 4: overloading the + operator for vectors----
41  vec operator+(const vec& a, const vec& b) { return sum(a, b); }
42  //----------------------------------------------------
43  // Optional Subtask 6: overloading the << operators-----
44  std::ostream& operator<<(std::ostream& os, const vec& v) {
45    os << "(" << v.x << "," << v.y << "," << v.z << ")";
46    return os;
47  }
48  std::ostream& operator<<(std::ostream& os, const line& l) {
49    os << l.start << " <-> " << l.end;
50    return os;
51  }
52  //----------------------------------------------------
```

≙ std::cout

cout

Function Overloading

Der Compiler muss immer zwischen *allen* Funktionen unterscheiden können. D.h. immer wenn Ihr eine Funktion benutzt, muss sie eindeutig identifizierbar sein.

Function prototype

Function prototypes include the function signature (number and types of the parameters), the name of the function, return type and access specifier (e.g. public or private in a class). (Wiki)

Function overloading heisst: Es existieren mehrere Funktionen mit dem gleichen Namen.

Function Overloading

Unterscheidung bzgw. eindeutige Bestimmung verschiedener Funktionen gleichen Namens

Funktionen werden eindeutig bestimmt durch:

- den Namen der Funktion
- die Anzahl der Argumente
- die Datentypen der Argumente
- die Reihenfolge der Argumente

f(int a,)
f(int a, int b)
f(float a)
f(

Nicht unterschieden wird durch:

- den Datentyp des return value
- die Namen der Argumente

f(int b)

Function Overloading

Beispiele

```
int fun1(const int a) { ... }
int fun1(const int a, const int b) { ... }

int fun2(const int a) { ... }
int fun2(const float a) { ... }

int fun3(const int a) { ... }
int fun3(const int b) { ... }

int fun4(const int a) { ... }
double fun4(const int a) { ... }
```

OK #Arg.

OK Typ.

nicht OK

nicht OK

Function Overloading

Beispiele

```
void out(const int i) {
  std::cout << i << " (int)\n";
}
void out(const double i) {
  std::cout << i << " (double)\n";
}
int main() {
  // setup ...
  out(3.5);
  out(2);
  out(2.0);
  out(0);
  out(0.0);
  return 0;
}
```

Function Overloading

Beispiele

```
void out(const int i) {
  std::cout << i << " (int)\n";
}
void out(const double i) {
  std::cout << i << " (double)\n";
}
int main() {
  // setup ...
  out(3.5);              // 3.5 (double)
  out(2);                // 2 (int)
  out(2.0);              // 2 (double)
  out(0);                // 0 (int)
  out(0.0);              // 0 (double)
  return 0;
}
```

Tribool

true
false
u

NOT(A)

A	¬A
F	T
U	U
T	F

AND(A,B)

A ∧B		B		
		F	U	T
A	F	F	F	F
	U	F	U	U
	T	F	U	T

OR(A,B)

A∨ B		B		
		F	U	T
A	F	F	U	T
	U	U	U	T
	T	T	T	T

F = FALSE, U = UNKNOWN, T = TRUE

- Wie könnten wir dieses Konzept intern implementieren?
- Was für Operationen und Funktionen wären nützlich?

Tribool

Was brauchen wir alles für eine eigene Klasse?

```
class Tribool {
private:
    // 0 means false, 1 means unknown, 2 means true.
    unsigned int value;  // INV: value in {0, 1, 2}.
public:
    // ...    set_value (int v){
                  assert(v ∈ {0, 1, 2})
                  value = v
};
```

- Was heisst private/public?
- Wieso wollen wir, dass value private ist?

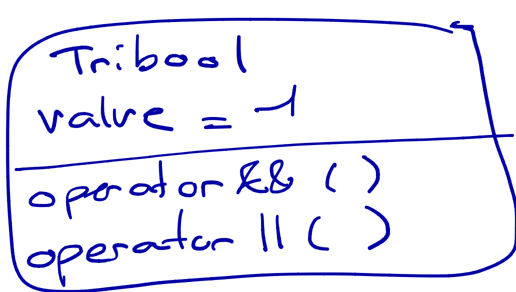
Tribool

Was brauchen wir alles für eine eigene Klasse?

- Was heisst private/public? Regelt die Sichtbarkeit der Variablen oder Funktionen. Heisst: auf private Sachen könnt ihr von ausserhalb der Klasse nicht via "." Operator zugreiffen.
- Wieso wollen wir, dass value private ist? Wie wir Tribool intern implementieren ist für die Funktion der Klasse egal, vlt. wollen wir später die interne Representation ändern. Weil wir value private gemacht haben, könnten wir alles intern ändern ohne, dass von "aussen" etwas bemerkbar gewesen wäre. Ausserdem können wir dann immer kontrollieren, dass value $\in \{0, 1, 2\}$ ist.

Constructors

Was brauchen wir alles für eine eigene Klasse?



= "a"

ist Teil der Klasse

A constructor in C++ is a special 'MEMBER FUNCTION' having the same name as that of its class which is used to initialize some valid values to the data members of an object. It is executed automatically whenever an object of a class is created. (Great Learning: Constructor in C++ and Types of Constructors)

```
Tribool a(1);
```

```
Tribool(uint v){
    // check invariant
    value = v;
}
```

Wo schreiben wir alles hin?

- Deklarationen schreiben wir in class_name**.h**
- Definitionen schreiben wir in class_name**.cpp**

Diese separation von Deklaration und Definition heisst "out-of-class definition".

- Deklarationen: Prototypen der Funktionen, sagt dem Compiler, diese Funktion existiert irgendwo.
- Definitionen: Die tatsächliche Implementierung dessen, was die Funktion ausführen soll.

Scope resolution operator ::

std::cout

Wenn wir in class_name**.cpp** die Funktionsdefinitionen implementieren, müssen wir dem Compiler irgendwie sagen, dass diese Funktion zu einer Klasse gehört. Das können wir mit dem :: erreichen.

```
// In tribool.h: Hier brauchen wir keine scope resolution,
// da wir schon innerhalb der Klasse sind.
std::string string() const;

// In tribool.cpp: Hier müssen wir anzeigen, dass diese Funktion
// zu der Klasse Tribool gehört.
std::string Tribool::string() const {...}
```

gehört zu dieser Klasse

Name der Fkt.

Unsere Aufgabe:

```
class Tribool {
private:
    // ...
public:
    // Constructor 1 (passing a numerical value)
    // PRE: value in {0, 1, 2}.
    // POST: tribool false if value was 0, unknown if 1, and true if 2.
    Tribool(unsigned int value_int);

    // Constructor 2 (passing a string value)
    // PRE: value in {"true", "false", "unknown"}.
    // POST: tribool false, true or unknown according to the input.
    Tribool(std::string value_str);

    // Member function string()
    // POST: Return the value as string
    std::string string() const;

    // Operator && overloading
    // POST: returns this AND other
    Tribool operator&&(const Tribool& other) const;
};
```

Const Member-Funktionen

in irgendeiner Klasse {

```
bool do_something(int input);

// return type is const
const bool do_something(int input);

// input is const
bool do_something(const int input);

// object that this function is called on is const
bool do_something(int input) const;
```

print(...) const;

value = input

}

const Tribool T(1);

T.do_something(5);

Initializer Lists

```
class Base
{
  private:
  int value;
  bool blabla;
  public:
  // default constructor
  Base(int v, bool b):value(v), blabla(b){...}
  // same as above, choose one
  Base(int v, bool b){
    value = v;
    blabla = b;
  }
};
```

gleicher Name, kein return type ⇒ Constr.

function body

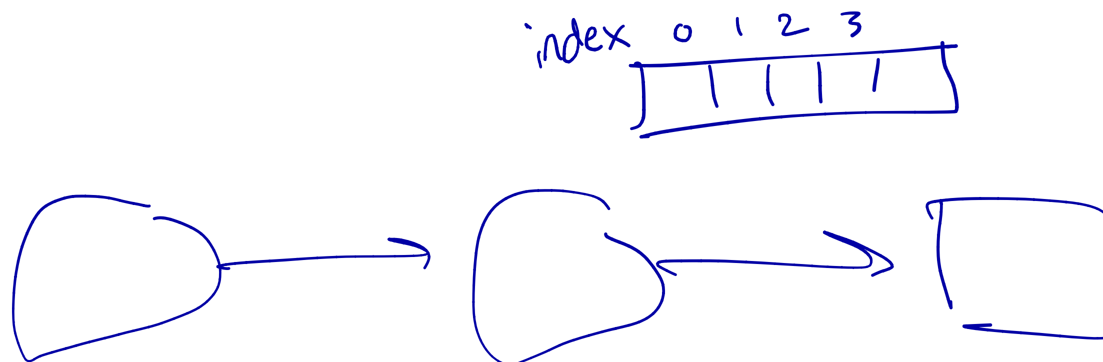
Was haben wir in dieser Aufgabe gesehen?

- Klassen und Structs
- Sichtbarkeit/Visibility
- Operator Overloading
- Deklaration vs Definition
- Out-of-Class Definitionen
- `const` Member-Funktionen
- Konstruktoren
- Member Initializer Listen

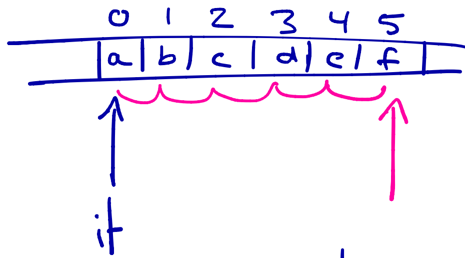
Containers in C++

Container sind Objekte um eine Sammlung an Elementen zu speichern. Einige Beispiele sind `std::vector`, `std::set`, `std::list`. Hier die vollständige Liste aller im standard library vorhandenen Container:
`https://en.cppreference.com/w/cpp/container`.

Um durch diese Container zu iterieren, stellen all diese Container sog. iterators/Iteratoren zu Verfügung. Hier brauchen wir als User nicht zu wissen wie die iterators intern implementiert sind.



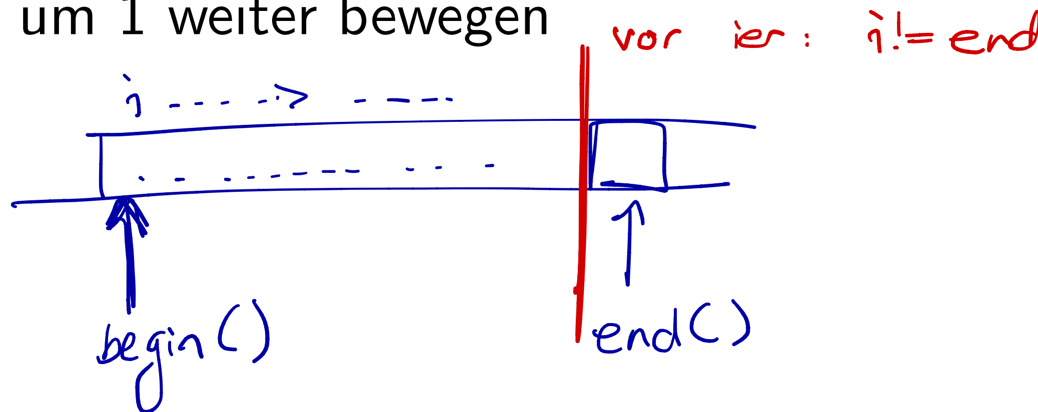
Iterators



std::vector< int > c ...
std::set ..

Für einen Container c haben wir:

- `it_Anfang` = c.begin(): Iterator zeigt auf das erste Element
- `it_Ende` = c.end(): Iterator zeigt hinter das letzte Element
- `*it`: Zugriff auf das jetzige Element (∼ wie vec[i])
- `++it`: Iterator um 1 weiter bewegen



iterator i = begin

Iterators

Beispiele (aus Summary 10)

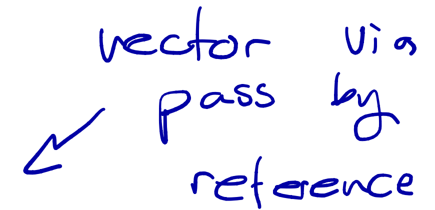
```
std::cout << "Enter 6 numbers:\n";
std::vector<int> a (6, 0);
for (std::vector<int>::iterator i = a.begin(); i < a.end(); ++i)
    std::cin >> *i; // write into object of iterator

for (std::vector<int>::const_iterator i = a.begin(); i < a.end(); ++i)
    std::cout << *i << std::endl; // read from object of iterator
}
```

noch besser: !=

Iterators

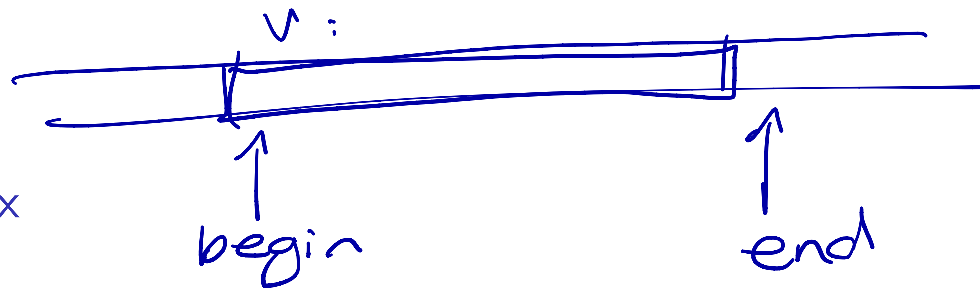
Code example: Find max

```
// PRE:  i < j <= v.size()
// POST: Returns the greatest element of all elements
// with indices between i and j (excluding j)
unsigned int find_max(const std::vector<unsigned int>& v,
                      unsigned int i,
                      unsigned int j)
{
  unsigned int max_value = 0;
  for (; i < j; ++i) {
    if (max_value < v.at(i)) {
      max_value = v.at(i);
    }
  }
  return max_value;
}
```



Iterators

Code example: Find max




```
// PRE:  (begin < end) && (begin and end must be valid iterators)
// POST: Return the greatest element in the range [begin, end)
unsigned int find_max(std::vector<unsigned int>::iterator begin,
                      std::vector<unsigned int>::iterator end)
{
  unsigned int max_value = 0;
  for(; begin != end; ++begin) {
    if (max_value < *begin) {
      max_value = *begin;
    }
  }
  return max_value;
}
```

vector als "Bereich zwischen zwei Iteratoren"

*iterator heisst immer auf Element/Wert zugreiffen, auf den der Iterator gerade zeigt.

Hinweis für die Serie: instream

e.g. valid input : [2,5]

```
#include "complex.h"

bool read_input(std::istream& in, Complex& a){
  unsigned char c;
   if(!(in >> c) || c != '['
      /* TODO: Finish implementation */
      || !(in >> c) || c != ']')
      return false;
    else
      return true;
}

std::istream& operator>>(std::istream& in, Complex& a) {
    if(!read_input(in, a)) {
        in.setstate(std::ios::failbit);
    }
    return in;
}


// TODO: Write all the operators and functions implementations here.
```

in hat [2,5] "drin"

// 2,5]

nach !(in >> a.real) hat in noch ,5]

→ Komma einlesen (wie ']')

→ img. einlesen

Algorithm Library

```
// POST: Reads a sequence terminated by a negative number into vec.
void input(std::vector<int>& vec) {...}
// POST: Prints the contents of the vector into stdout.
void print(std::vector<int>& vec) {...}

int main() {
  std::vector<int> vec = std::vector<int>();
  input(vec);

  int largest = *std::max_element(vec.begin(), vec.end());
  std::cout << "Largest element is: " << largest << std::endl;

  std::sort(vec.begin(), vec.end());
  std::cout << "Sorted vector: ";
  print(vec);
}
```

Wir kriegen einen iterator zurück, wollen aber das Element

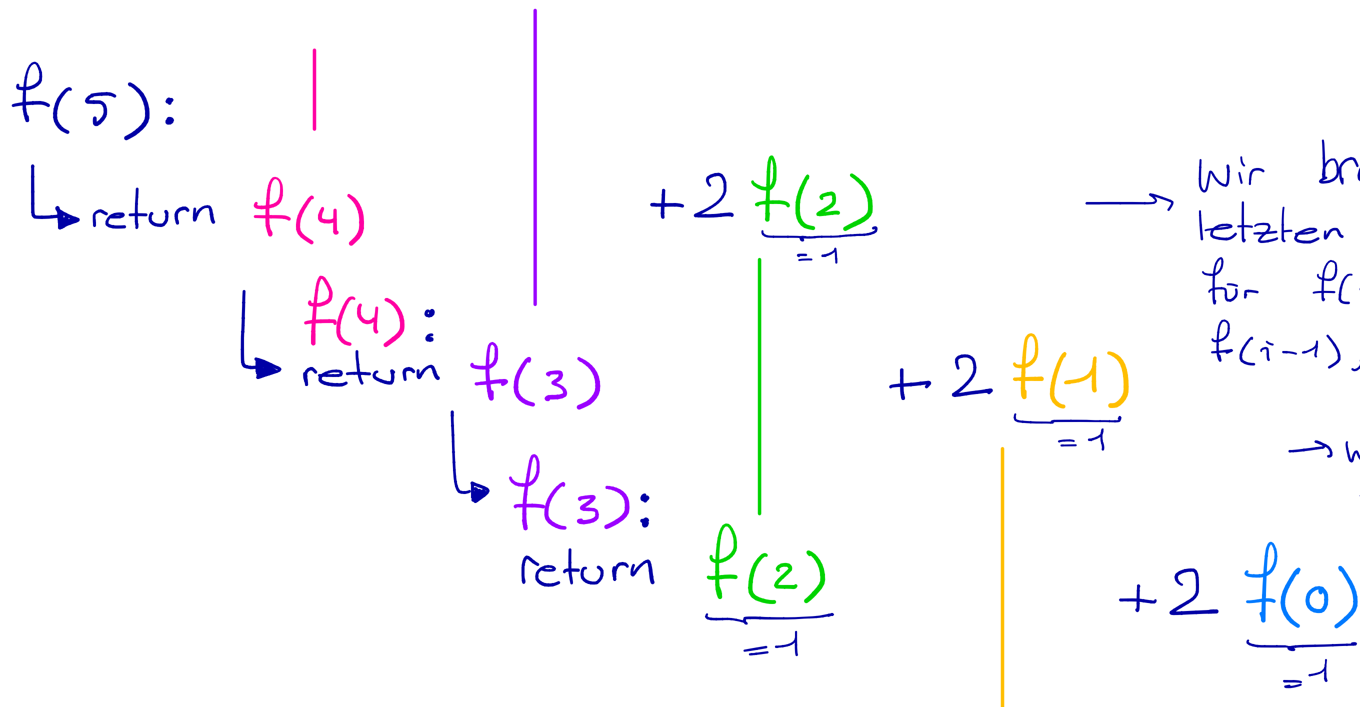
Wir passen [begin, end[an die Fkt

Recursion to Iteration

Siehe Lecture 10: Exercise Session (code Example)

→ Die meisten (probs fast alle) Rekursiven Codes kann man auch iterativ schreiben. Hier einfach ein Bsp. wie man vorgehen könnte.

```
unsigned int f(const unsigned int n) {
  if (n <= 2) return 1;
  return f(n-1) + 2 * f(n-3);
}
```



→ Wir brauchen immer die letzten 3 Aufrufe d.h. für $f(i)$ brauchen wir $f(i-1)$, $f(i-2)$, $f(i-3)$

→ wobei wir nur $f(i-1)$ und $f(i-3)$ benutzen.

Recursion to Iteration

Siehe Lecture 10: Exercise Session

```
unsigned int f_it(const unsigned int n) {
    if (n <= 2)
        return 1;


    unsigned int f2 = 1;    // f(0) = f(i-2) für i==2
    unsigned int f1 = 1;    // f(1) = f(i-1) für i==2
    unsigned int f = 1;     // f(2) = f(i-0) für i==2

    for (unsigned int i = 3; i < n; ++i) {
        const unsigned int f3 = f2;  // f(i-3)
        f2 = b;                      // f(i-2)
        f1 = c;                      // f(i-1)
        f = f1 + 2 * f3;             // f(i)
    }

    return f + 2 * f2
}
```

Handwritten notes:

- Next to the loop body: hier berechnen wir immer die letzten 3 f(·) Werte, wie oben.
- Below the loop: $f(i) = f(i-1) + 2 \cdot f(i-3)$
- Below the return statement: $f(i+1) + 2 \cdot f(i-2)$, where $f(i-2) = f((i+1)-3)$
- Next to the return statement: hier in der letzten Ausführung berechnen wir quasi $f(i+1)$, weil die letzte loop iteration ja schon $f(i)$ ausgerechnet hat.