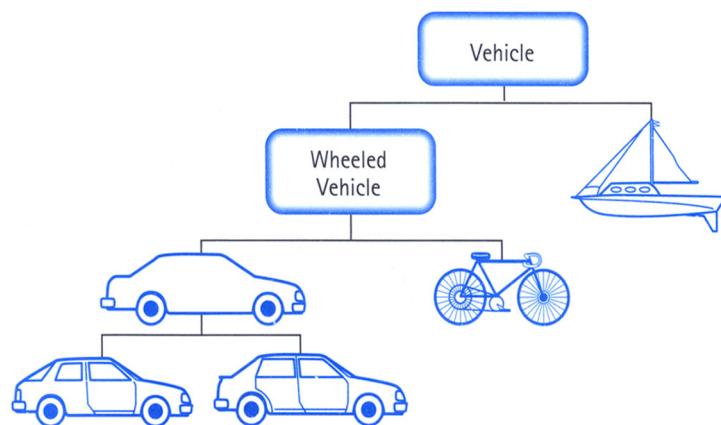




Objekt-Orientierte Programmierung

VORLESUNG 5





Overview

- Wiederholung
- Modellierung
- Vorlagen (Templates)
- Standardbibliothek



**TRY BY
YOURSELF**

**SEARCH ON
STACK OVERFLOW**

READ THE DOCS

**READ THE
SOURCE CODE**

Level of programming



Vom Quellcode zum Programm 1

Eine Toolchain vom Quellcode zum ausführbaren Programm:
eingeben, übersetzen und binden.

1. Editor

- Ein- und Ausgabe, und Speicherung, des Quellcodes
- Ergebnis: Quellcodedateien (etwa myProg.cpp, library.h)

2. Präprozessor (oft integrierter Teil des Compilers)

- Ergänzungen und Ersetzungen innerhalb des C++ Quellcodes
- Ergebnis: Quellcode als Input für die Übersetzung durch den Compiler



Vom Quellcode zum Programm 2

Eine Toolchain vom Quellcode zum ausführbaren Programm:
eingeben, übersetzen und binden.

3. Compiler

- Übersetzung der präprozessierten C++ Quellcode-Datei(en) in binären Objektcode
- Ergebnis: Objektcodedateien (etwa myProg.o, library.o) als Input für den Linker

4. Linker

- Zusammenführung der unterschiedlichen Objektcodes (etwa aus mehreren Objektcodedateien, insbesondere aus den verwendeten Bibliotheken)
- Ergebnis: ausführbare Datei (etwa myProg.exe)



Vom Quellcode zum Programm

Eine Toolchain vom Quellcode zum ausführbaren Programm:
eingeben, übersetzen und binden.

compile:

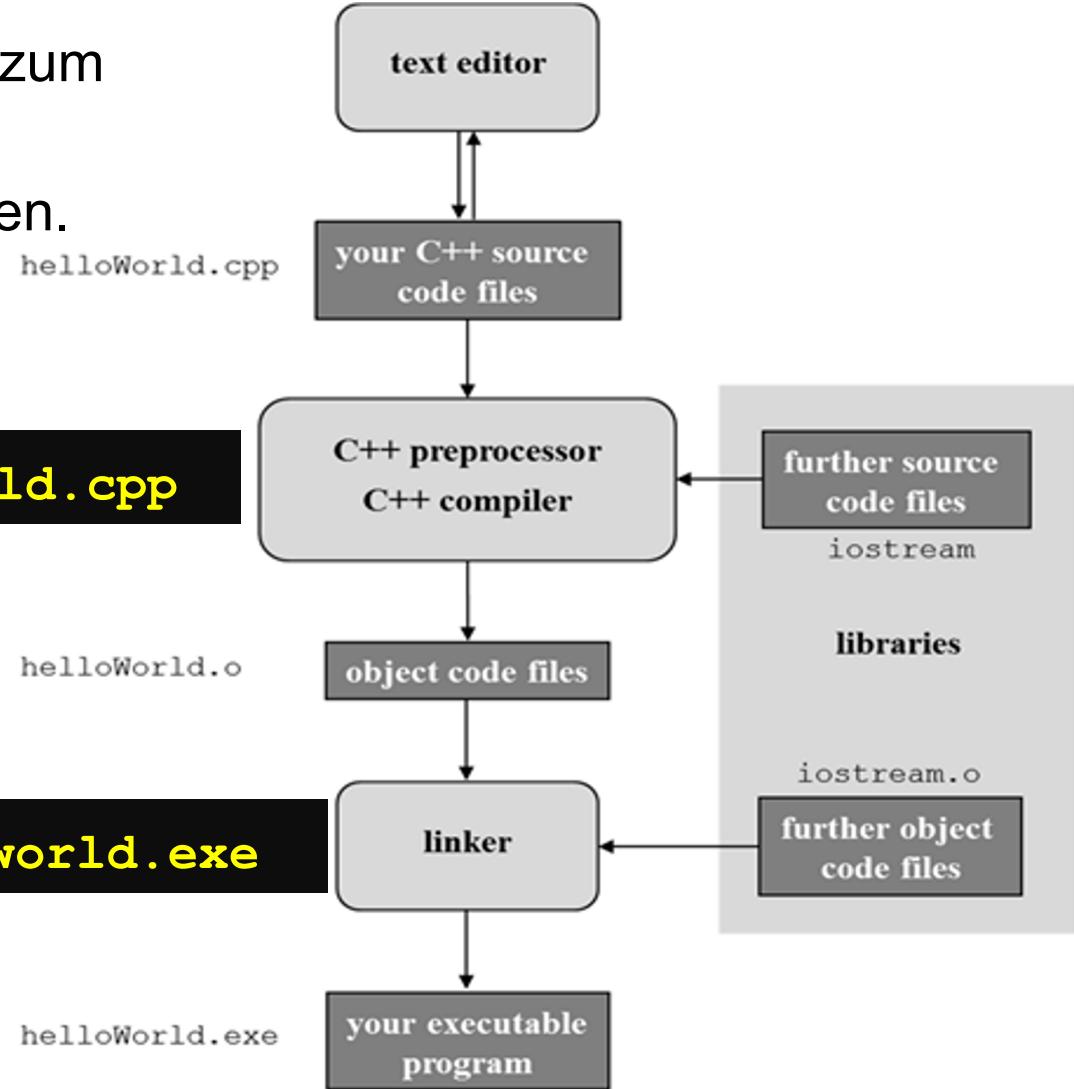
```
g++ -c helloworld.cpp
```

link:

```
g++ helloworld.o -o helloworld.exe
```

compile & link:

```
g++ helloworld.cpp -o helloworld.exe
```





Kompilieren von modularem Programm

Module können **unabhängig** kompiliert werden.

Module (1 oder mehr)

class_declaration.h enthält die Definition der Klasse

class_implementation.cpp enthält die Implementierung

g++ -c class_implementation.cpp → class_implementation.o

Mainprogram

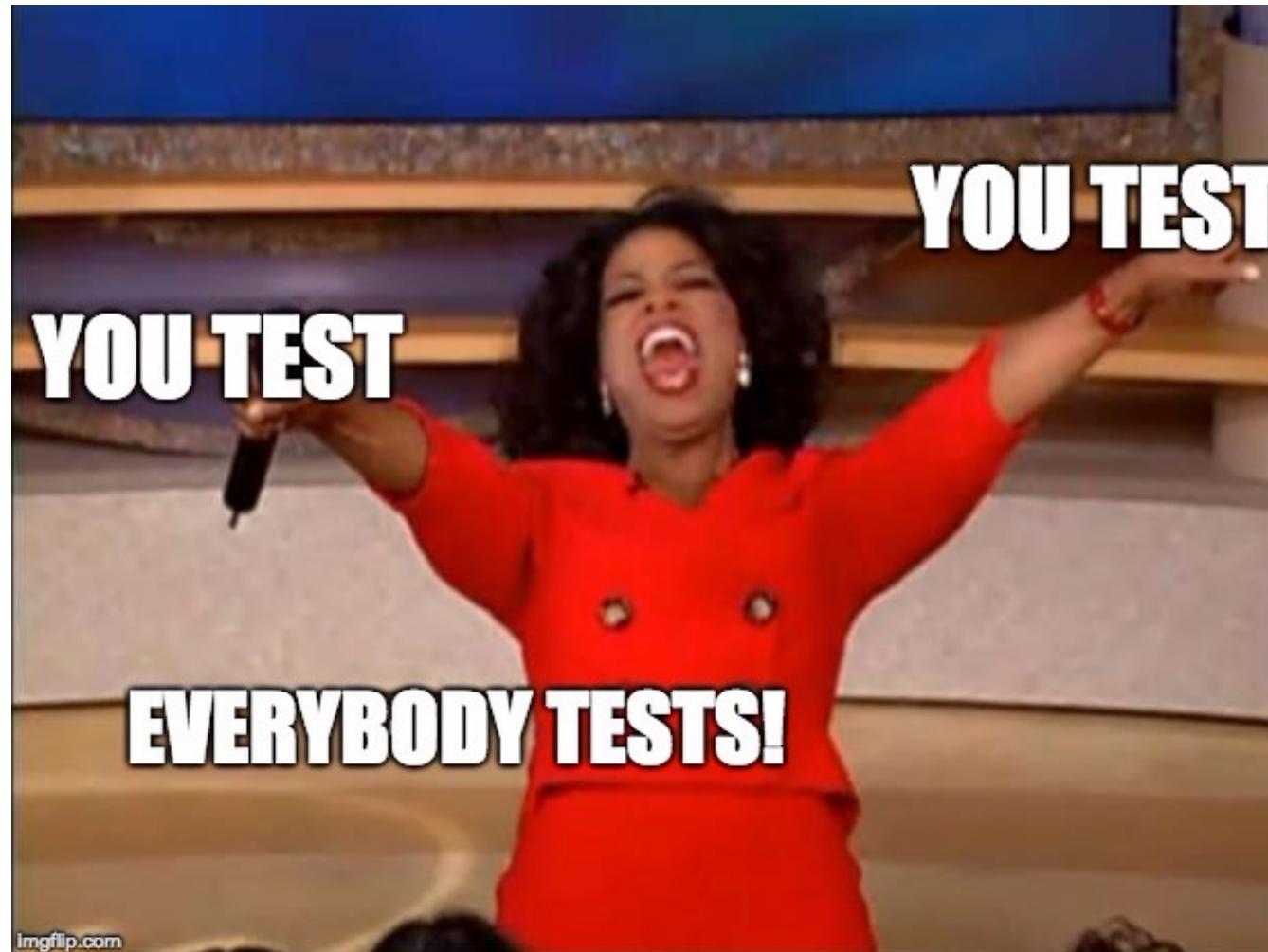
user_program.cpp enthält main() die die Klasse verwendet

g++ -c user_program.cpp → user_program.o

**g++ user_program.o class_implementation.o -o
user_program.exe**

Module können **gemeinsam** kompiliert werden.

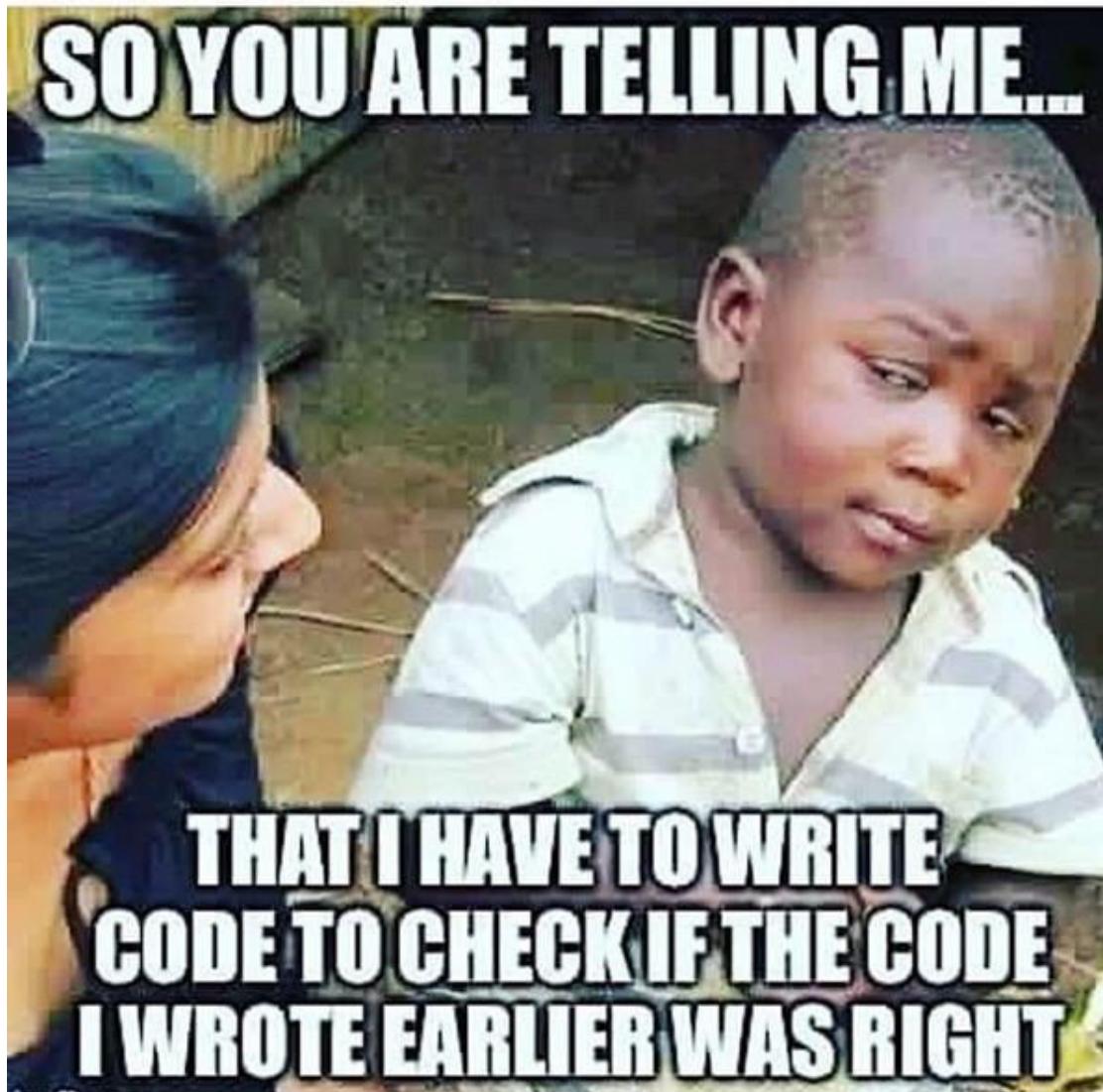
**g++ user_program.cpp class_implementation.cpp -o
user_program.exe**





Testen von Programmen

- A. Manuell** ... ein Mensch führt das Programm mit einem Test-Protokoll, macht vorgegebene Eingaben und überprüft die erwartete Ausgabe
 - B. Automatisch** ... ein Testprogramm führt das Programm aus, und simuliert Eingaben und verifiziert Ausgaben
-
- 1. Unit Testing** – Testprogramm für Funktionen, Klassen oder Module
 - 2. Integration Testing** – Test des Zusammenwirkens verschiedener Module
 - 3. Functional Testing** – Test der Benutzeroberfläche und der gesamten Programmlogik
-
- Robustheit eines Programms wird in der Praxis durch Tests überprüft.
 - Tests müssen widerholbar sein.
 - Sowohl 1., 2., als auch 3. kann automatisiert werden!





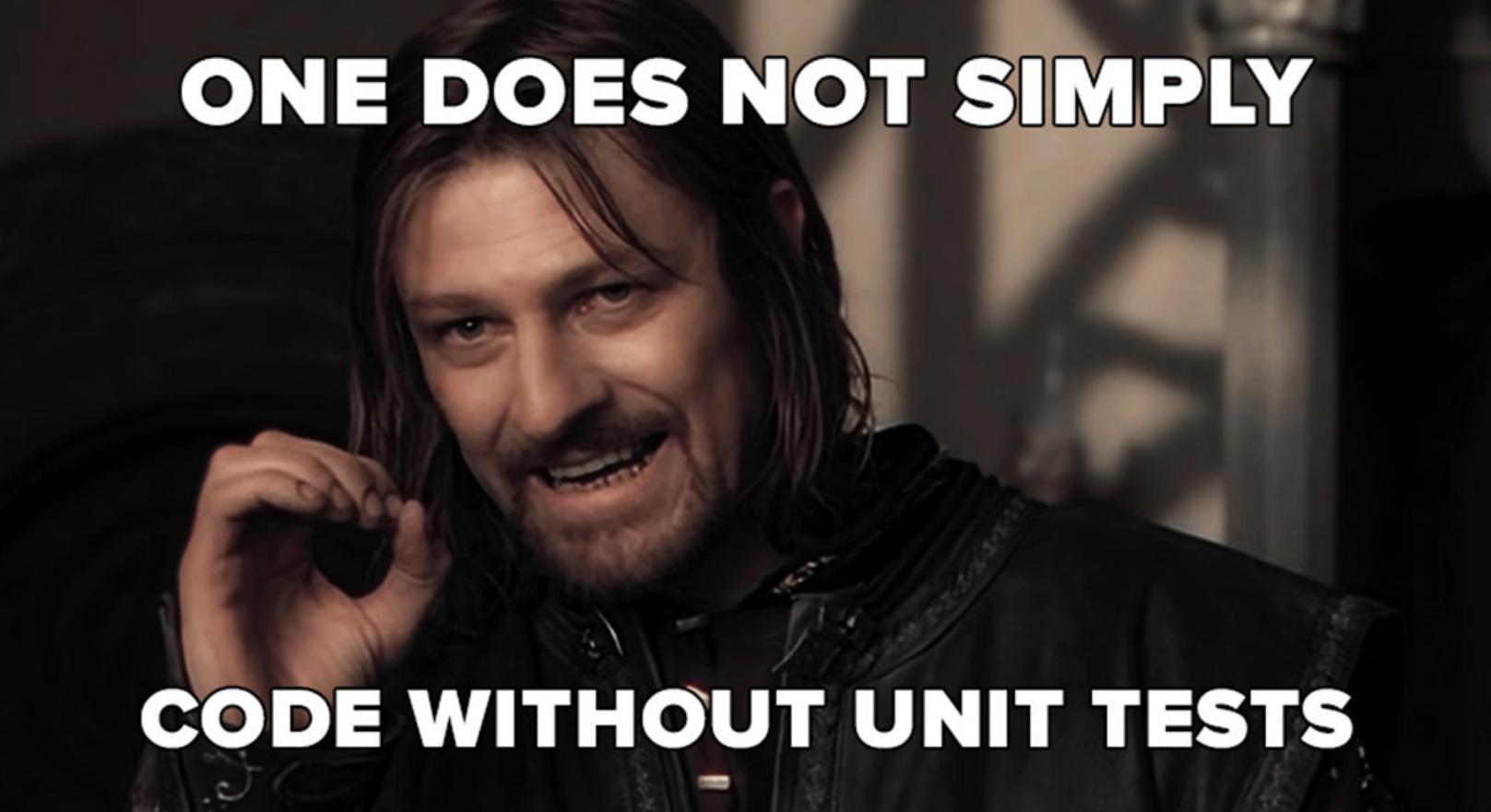
Automatisches Testen von Funktionen - Unit testing

- `#include <cassert>`
- `void assert (int expr);`
- wenn der Ausdruck auf 0 ausgewertet wird, wird eine Nachricht auf das Standardfehler geschrieben und **die Ausführung wird gestoppt**
- die Nachricht enthält: den Ausdruck, dessen Zusicherung fehlgeschlagen ist, den Namen der Quelldatei und die Zeilennummer, in der sie aufgetreten ist.

```
int main() {  
    ...;  
    int result = my_function(2,3);  
  
    assert(result==5) clang version 7.0.0-3~ubuntu0.18.04.1 (tags/RELEASE_700/final)  
}                                     ▶ clang++-7 -pthread -o main main.cpp  
                                         ▶ ./main  
main: main.cpp:44: int main(): Assertion `2+2==5' failed.  
exited, aborted
```



ONE DOES NOT SIMPLY



CODE WITHOUT UNIT TESTS



Modellierung

Unified Modelling Language

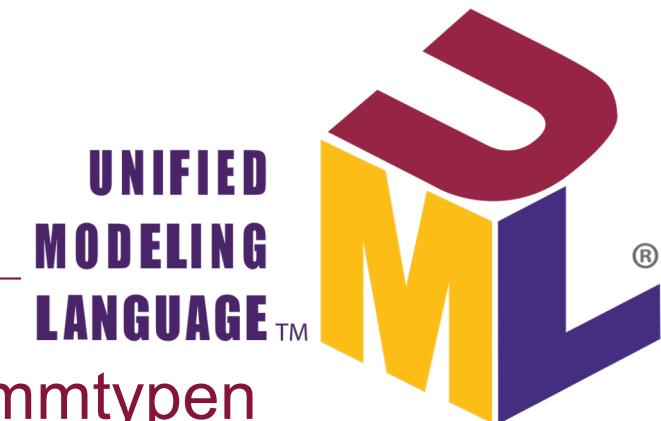


- **Standardsprache** in Industrie zum Modellieren:
Spezifizieren, Visualisieren und Dokumentieren von
Softwaresystemen.
- Die Standard-Notation für Softwarearchitektur.
- UML ist Programmiersprach-**unabhängig**

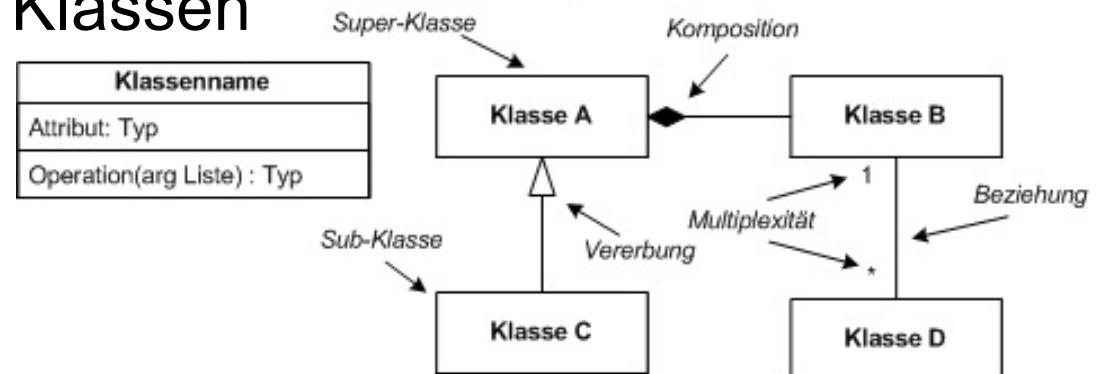
Tools:

Lucidchart (online), ArgoUML, Visio (Microsoft), Rational
Software Modeler (IBM) ...

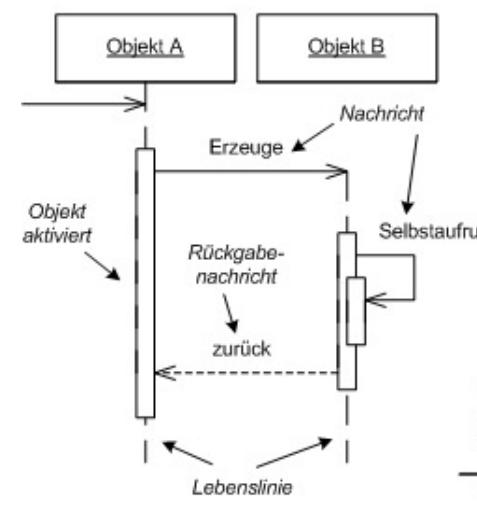
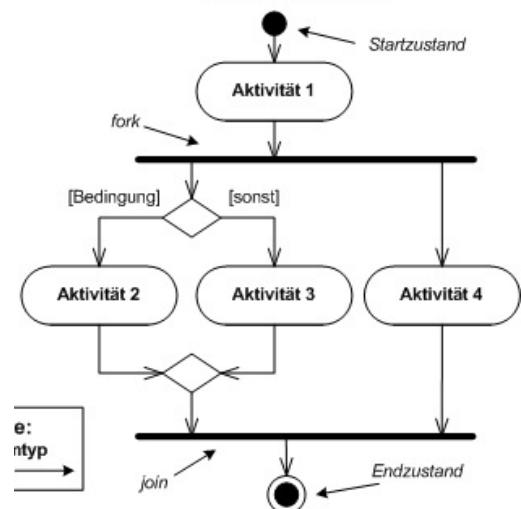
Unified Modelling Language



- UML beinhaltet verschiedene **Diagrammtypen**
- Strukturdiagramme, ex. Klassen



- Verhaltensdiagramme, ex. Aktivitäten und Sequenzen

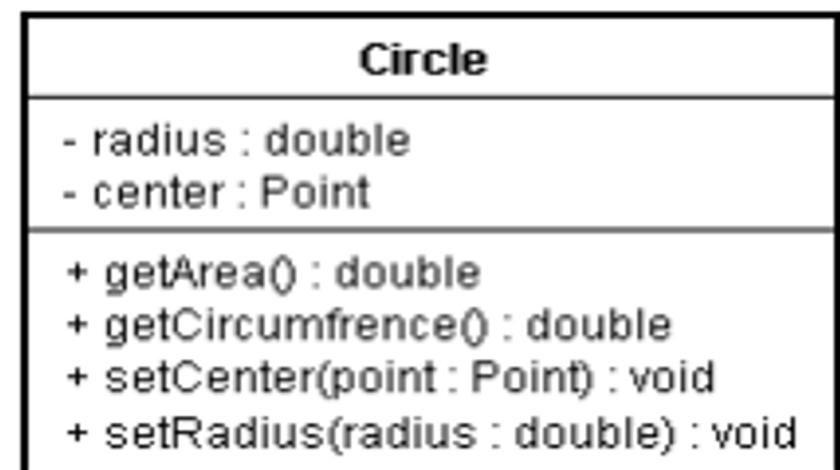




Klassendiagramme

- Ein UML-Klassendiagramm spezifiziert die Entitäten in einem Programm und die Beziehungen zwischen ihnen.
- Es enthält und spezifiziert:
 - Klassenname
 - Attribute (Name, Typ)
 - Methoden (Name, Parametertypen, Rückgabetypr)

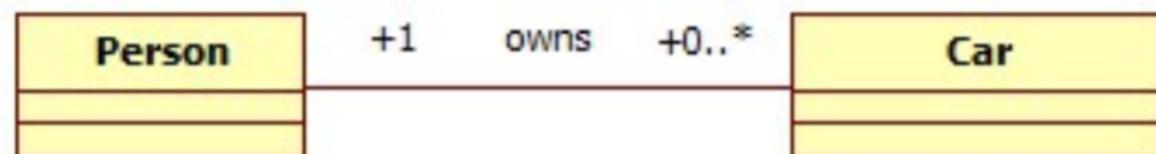
- private Members: -
- public Members: +
- protected Members: #





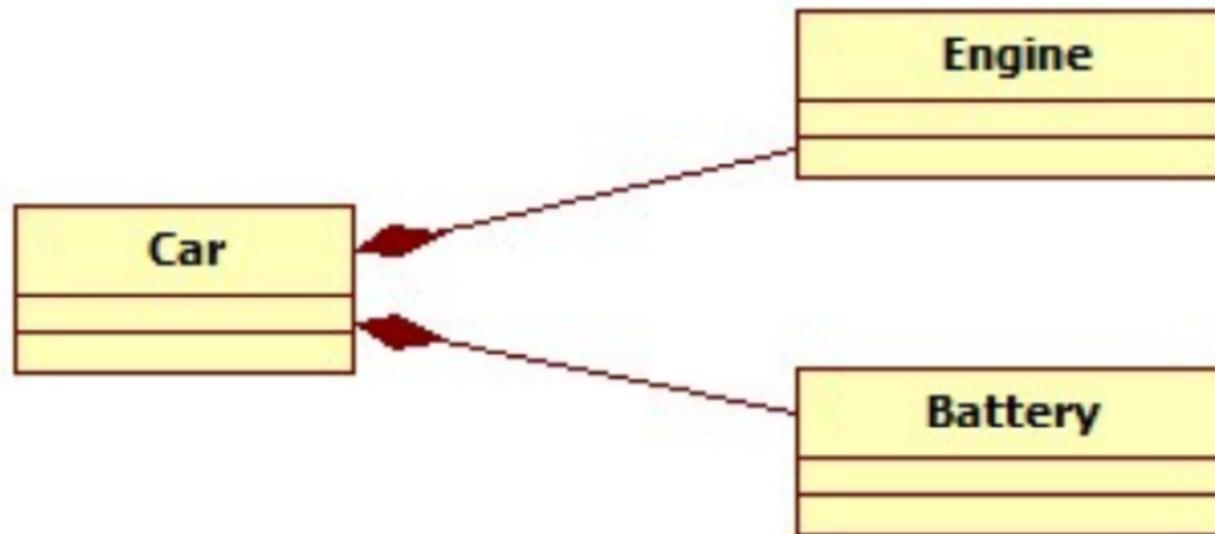
Assoziation I

- UML-Assoziationen beschreiben Beziehungen struktureller Abhangigkeit zwischen den Klassen.
- Ein Assoziation kann haben:
 - Name;
 - Kardinalitat;
 - Navigierbarkeit (uni / bidirektional).
- **Assoziation** (knows-a) - ist eine Beziehung zwischen zwei Klassen. Eine Klasse A enthalt eine Referenz zu einer anderen Klasse B.



Assoziation II

Komposition (has-a) - wenn Klasse B ist ein Teil von Klasse A, steuert die Klasse A die Lebensdauer von der Instanz der Klasse B. Diese Instanz wird zerstört, wenn die Instanz der Klasse A zerstört wird.



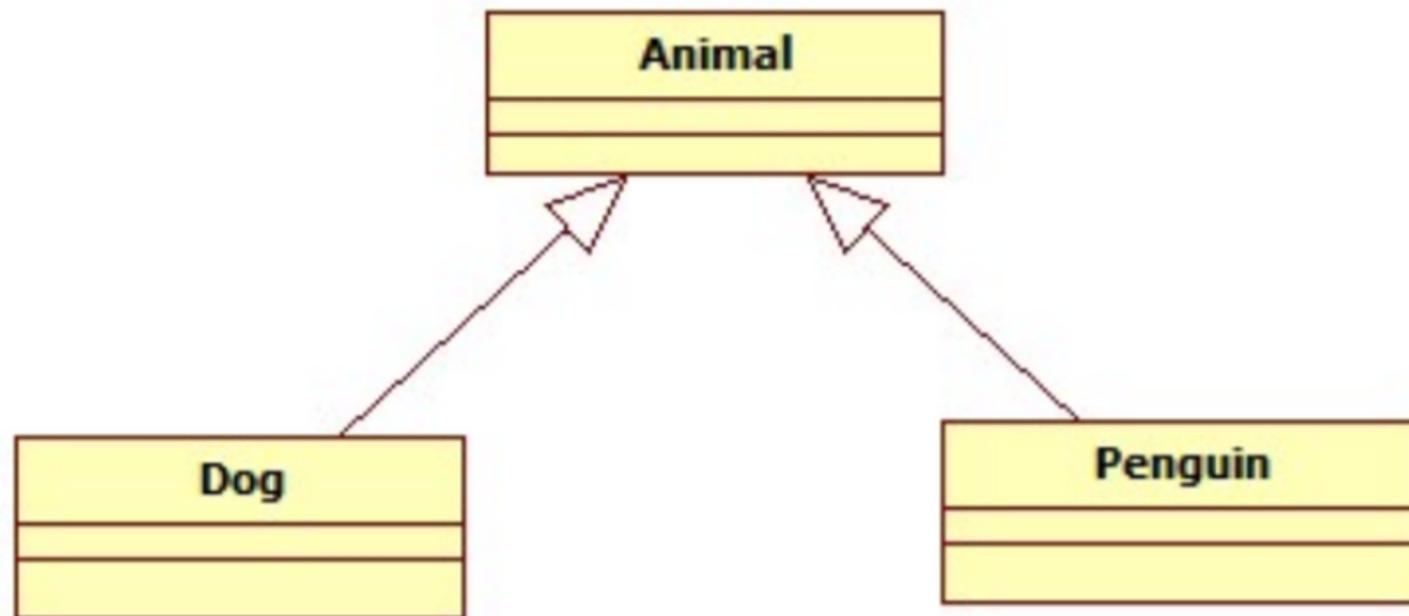
Assoziation III

Abhängigkeit/Dependency (uses-a) - wenn Klasse A in einer Methode einen Referenz auf Klasse B verwendet (Parameter oder lokal Variable). Eine Änderung in der Klasse B kann A beeinflussen.



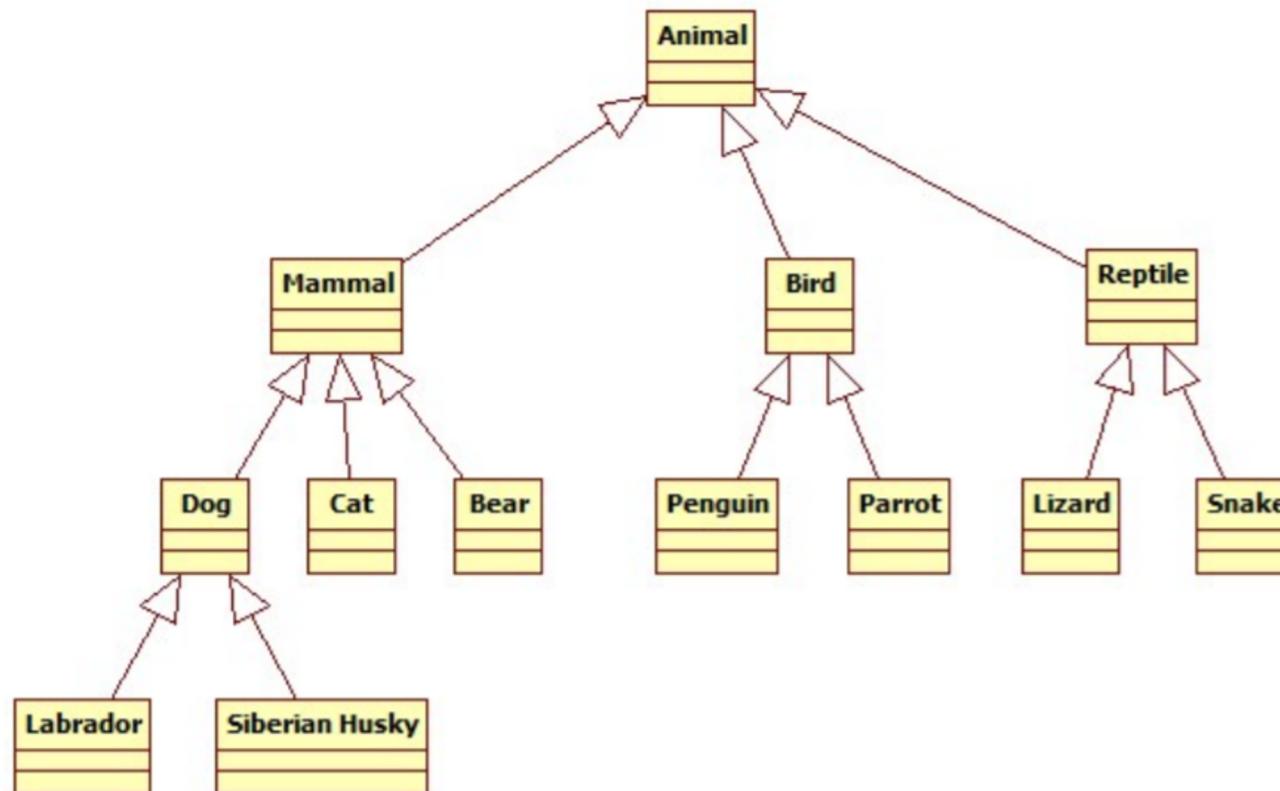
Assoziation IV

Vererbung (is-a) - jede Instanz der abgeleiteten Klasse ist eine Instanz der Basisklasse.



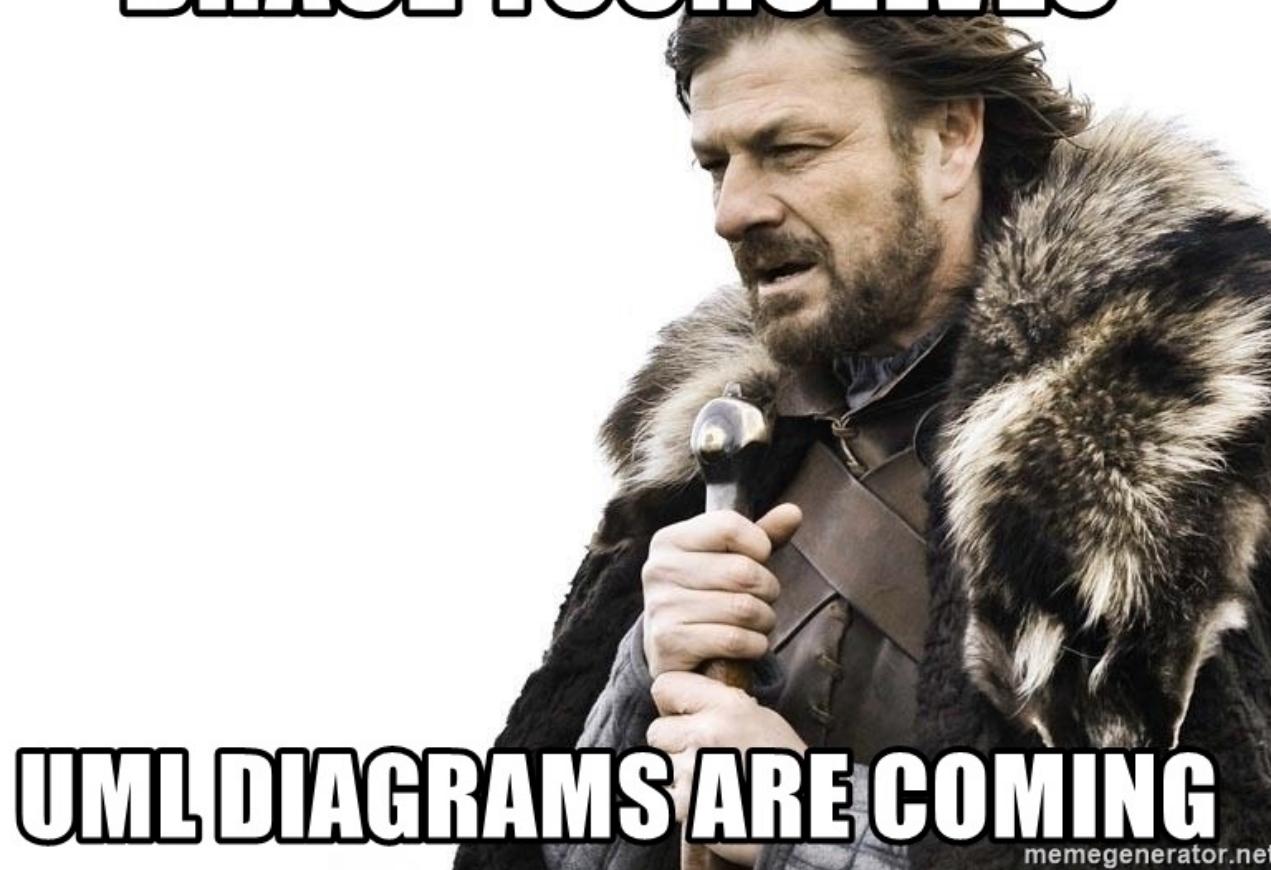
Assoziation V

Vererbung erlaubt die Definition der Hierarchien von verwandten Klassen.





BRACE YOURSELVES



UML DIAGRAMS ARE COMING

memegenerator.net



Schablonen (Templates)





Templates

- erlauben Arbeit mit generischen Typen
- bieten Möglichkeit zur Wiederverwendung von Code
 - der Code ist nur einmal geschrieben und er kann dann mit vielen verschiedenen Typen verwendet sein
- erlaubt eine Funktion oder eine Klasse mit verschiedenen Typen zu arbeiten
 - es ist mit verschiedenen Typen parametrisiert



Funktion Templates I

Deklaration

```
template <type_name identifier> function_declaration;
```

```
template <typename T>
T add(T a, T b)
{
    return a + b;
}
```

- **T** ist der Template-Parameter: ein Typargument für das Template
- der Template-Parameter kann mit einem der folgenden zwei Schlüsselwörter benutzt werden: **typename**, **class**



Funktion Templates II

Der Prozess zum Generieren einer effektive Funktion aus einer Template-Deklaration heißt **Instanziierung (Instantiation)**:

```
int resInt = add<int>(3, 4);
double resDouble = add<double>(-1.2, 2.6);
```



Klassen Templates I

- ein Template kann wie ein Makro aufgefasst werden
- beim Instanziieren eines Templates erstellt der Compiler eine neue Klasse mit den angegebenen Typ-Argumenten
- der Compiler benötigt Zugriff auf die Implementierung der Methoden, um das Template zu instanziieren
- die Definition eines Templates erfolgt in einer Header-Datei



Klassen Templates II

Templates können auch für mehr Typen definiert werden:

```
template <typename T, typename U>
class Pair
{
private:
    T first;
    U second;
// ...
};
```



Klassen Templates Beispiel

```
// A class to represent a stack
template <class X>
class stack
{
    X *arr;
    int top;
    int capacity;

public:
    stack(int size) {
        arr = new X[size];
        capacity = size;
        top = -1;
    }

    ~stack() {
        delete[] arr;
    }

    void push(X x) {
        if (isFull()) Raise_Error("Stack Full");
        arr[++top] = x;
    }

    X pop() {
        if (isEmpty()) Raise_Error("Empty Stack");
        return arr[top--];
    }

    X peek() {
        if (isEmpty()) Raise_Error("Empty Stack");
        else return arr[top];
    }

    int size() {
        return top + 1;
    }

    bool isEmpty() {
        return top == -1;
    }

    bool isFull() {
        return top == capacity - 1;
    }
};
```



Klassen Templates Beispiel

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    stack<string> pt(2);

    pt.push("A");
    pt.push("B");

    pt.pop();
    pt.pop();

    pt.push("C");

    // Prints the top of the stack
    cout << "The top element is " <<
    pt.peek() << endl;

    // Returns the total number of
    // elements present in the stack
    cout << "The stack size is " <<
    pt.size() << endl;

    pt.pop();

    // check if the stack is empty or not
    if (pt.isEmpty()) {
        cout << "The stack is empty\n";
    }
    else {
        cout << "The stack is not
empty\n";
    }

    return 0;
}
```

Standard Template Library (STL)





Standard Template Library (STL)

- ist Teil der Standardbibliothek für C ++
- die Komponenten der STL sind Templates
- damit kann der Programmierer Komponenten ohne Leistungsverlust erstellen
- der primäre Entwickler von STL ist Alexander Alexandrowitsch Stepanow



Container in STL I

- ein Container ist ein Objekt, das eine Sammlung von anderen Objekten (seine Elemente) enthält
- Container werden als Templates implementiert
- Container:
 - verwalten den Speicherplatz für Elemente;
 - haben Methoden zum Zugriff auf die Elemente (direkt oder über Iteratoren);
 - stellen Methoden zum Ändern der Elemente bereit.



Containers in STL II

- **Sequence Containers** (Elemente sind linear angeordnet):
 - `array<T>;`
 - `vector<T>;`
 - `deque<T>;`
 - `forward_list<T>;`
 - `list<T>.`
- **Associative Containers** (Elemente werden durch Schlüssel zugegriffen):
 - `set<T, CompareT>;`
 - `multiset<T, CompareT>;`
 - `map<KeyT,ValueT,CompareT>;`
 - `multimap<KeyT, ValueT, CompareT>.`



Containers in STL III

- **Containers adapters** (Einschränkung der Funktionalität in einem vorhandenen Container):
 - `stack<T, ContainerT>;`
 - `queue<T, ContainerT>;`
 - `priority_queue<T,ContainerT, CompareT>.`



Iteratoren I

- Geben generische Möglichkeit, um auf die Elemente zuzugreifen
- Zugriff auf die Elemente ohne Annahmen bezüglich der internen Repräsentation (Implementation Hiding)
- Trennung: wie Daten gespeichert, wie bearbeitet werden
- Ein Iterator enthält:
 - ein **Reference** auf das aktuelle Element;
 - ein **Reference** auf den Container.



Iteratoren II

- Iteratoren bieten Methoden für **Traversierung** (Traversal), **dereferencing** und **Bound Detection**
- C++ **Iterators** sind keine **Pointers** (Zeiger)
 - Allerdings: der `++` Operatoren lässt den Iterator weiterwandern!
Und: der `*` Operator gibt das aktuelle Objekt
- Containers Methoden `begin()` und `end()` liefern Iteratoren



std::vector

- ist ein Container, der Elemente desselben Typs speichert
- ist ein **Sequence Container**
- abhängig von Bedarf ändert sich die Größe automatisch
- verwendet ein dynamisches Array, Elemente zu speichern
- der Elementzugriff ist sehr effizient (konstante/lineare Zeit)
- funktioniert mit range-based for-Schleife



std::vector

```
using namespace std;
int main()
{
    vector<int> vec = {1, 2, 3, 4, 5};
    int i;
    std::cout << "Please enter some integers (enter 0 to end):\n";
    do {
        std::cin >> i;
        vec.push_back (i);
    } while (i);

    // Displaying vector elements using begin() and end()
    cout << "The vector elements are : ";
    for (vector<int>::iterator itr = vec.begin(); itr < vec.end();
itr++) //moving iterator
        cout << *itr << " "; //element access
    return 0;
}
```



STL Algorithmen

- Headers: `<algorithm>` und `<numeric>`
- Algorithmen sind Function Templates, die auf Elementbereiche (**Ranges**) angewandt werden können.
 - Diese Ranges sind von Iteratoren definiert.
- Iteratoren, werden von den Funktionen `begin()` und `end()` zurückgegebenen sind.
- Iteratoren erlauben eine Trennung zwischen Algorithmen und Containern
 - man kann dieselbe Funktion auf verschiedene Containers anwenden (`find`, `sort`, `count` usw)



find

- `find(first, last, val)`
- sucht nach erstem Element im Bereich `[first, last[` gleich mit `val`

```
#include <iostream>
#include <algorithm>
#include <vector>

int main() {
    int n1 = 3;

    std::vector<int> v = {0, 1, 2, 3, 4};

    auto result1 = std::find(v.begin(), v.end(), n1);

    if (result1 != v.end()) {
        std::cout << "v contains: " << n1 << '\n';
    } else {
        std::cout << "v does not contain: " << n1 << '\n';
    }
}
```

Defined in header `<algorithm>`

```
template< class InputIt, class T >
InputIt find( InputIt first, InputIt last, const T& value );
```

(1)

```
template< class InputIt, class UnaryPredicate >
InputIt find_if( InputIt first, InputIt last,
                 UnaryPredicate p );
```

(2)

```
template< class InputIt, class UnaryPredicate >
InputIt find_if_not( InputIt first, InputIt last,
                     UnaryPredicate q );
```

(3)



sort

- `sort(first, last)`
- sortiert die Elemente im Bereich [first, last) in aufsteigender Reihenfolge (Compare)
- die Reihenfolge gleicher Elemente kann verändert werden

```
int main() {  
  
    std::vector<int> s = {5, 7, 4, 2, 8, 6, 1, 9, 0, 3};  
  
    std::sort(s.begin(), s.end());  
  
    for (int a : s) std::cout << a << " ";  
  
    std::cout << '\n';  
  
    std::sort(s.begin(), s.end(), std::greater<int>());  
  
    for (int a : s) std::cout << a << " ";  
  
    std::cout << '\n';  
}
```

Defined in header `<algorithm>`

`template< class RandomIt >
void sort(RandomIt first, RandomIt last);` (1)

`template< class RandomIt, class Compare >
void sort(RandomIt first, RandomIt last, Compare comp);` (2)



STL Algorithmen Vorteile

Einfachheit: man kann vorhandenen Code nutzen

Korrektheit: ist verifiziert

Leistung: Im Allgemeinen besser als selbst geschriebener Code

Klarheit: man kann die Intention einfach erkennen, z.B. einen `sort()` Aufruf Elemente in einem Range sortiert

Wartbarkeit: Code ist klar und einfach ⇒ einfacher zu schreiben, zu lesen, zu verbessern und zu warten



Fragen und Antworten