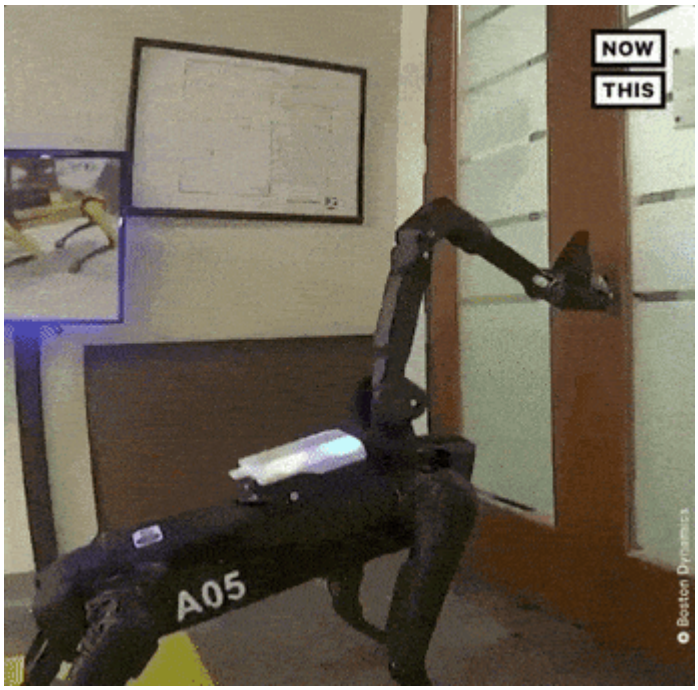
OOP und Container

Parameter	Kursinformationen
Veranstaltung:	`Softwareprojekt Robotik`
Semester	`Wintersemester 2022/23`
Hochschule:	`Technische Universität Freiberg`
Inhalte:	`Container in C++`
Link auf GitHub:	https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/04_OOP_Container.md
Autoren	Sebastian Zug & Georg Jäger



Zielstellung der heutigen Veranstaltung

- Zusammenfassung Objektorientierter Konzepte unter C++ (in Ergänzung zu Vorlesung 01)
  - Vererbungsmechanismen
  - Mehrfachvererbung
- knappe Einführung in die Container der Standardbibliothek
- Überblick zu den Algorithmen über den Containertypen

Fragen aus der vergangenen Woche

STL Kurzeinführung

Nehmen wir mal an ... sie wollen ein Projekt in C++ realisieren. Welche Unterstützung haben Sie von Seiten der Sprache, um einen wartbaren und übertragbaren Code zu definieren. Welchen Implementierungsmustern sollen Sie folgen:

- Verwendung der Standardbibliothek (kleiner Maßstab)
- Implementierung von Designpatterns (mittlere Ebene)
- externe Bibliotheken
- Realisierung von existierenden Architekturkonzepten

Anwendungsbeispiel

Ausgangspunkt ist die Behandlung von Datensammlungen, wie Sie sie aus C kennen. Wir erzeugen ein Array, innerhalb dessen die Datenstrukturen sequential abgelegt werden.

rawarray.cpp

```cpp
#include <iostream>

class Student{
  private:
    int matrikel;
    std::string name;
  public:
    Student(int matrikel, std::string name): matrikel(matrikel), nam
      ) {};
    friend std::ostream& operator<<(std::ostream& os, const Student&
};

std::ostream& operator<<(std::ostream& os,  const Student& s){
   os << s.name << "," << s.matrikel;
 return os;
}

int main()
{
  const int size = 3;
  Student studlist[] = {Student(123, "Alexander"),
                        Student(345, "Julius"),
                        Student(789, "Karl")};

  // Iteration mittels Indexvariable
  for(unsigned int i = 0; i < sizeof(studlist) / sizeof(class Student
    ++)
  {
     std::cout << &studlist[i] << " - " << studlist[i] << std::endl;
  }

  std::cout << std::endl;

  // Iteration über Größeninforamtion
  for (Student* i = studlist; i <= &studlist[size-1]; i++){
    std::cout << i << " - " <<  *i << std::endl;
  }

  return EXIT_SUCCESS;
}
```

```
0x7ffcebaef830 - Alexander,123
0x7ffcebaef858 - Julius,345
0x7ffcebaef880 - Karl,789

0x7ffcebaef830 - Alexander,123
0x7ffcebaef858 - Julius,345
0x7ffcebaef880 - Karl,789
```

Welche Nachteile ergeben sich aus diesem Lösungsansatz?

Wenn das Array an eine Funktion übergeben wird, verlieren wir die Information über seine Größe! Es fehlt letztendlich eine Managementebene für die Datensammlung. Zudem ist die statische Konfiguration in vielen Fällen hinderlich.

Lösungsansatz 1: Wir implementieren uns jeweils eine Management-Klasse, die die Aggregation des Speichers, die aktuelle Größe usw. für uns koordiniert.

Lösungsansatz 2: Wir verwenden die Standardbibliothek, die unter anderem generische Containertypen bereithält.

stdarray.cpp

```cpp
#include <iostream>
#include <array>

class Student{
  private:
    int matrikel;
    std::string name;
  public:
    Student(int matrikel, std::string name): matrikel(matrikel), nam
      ) {};
    friend std::ostream& operator<<(std::ostream& os, const Student&
};

std::ostream& operator<<(std::ostream& os,  const Student& s){
   os << s.name << "," << s.matrikel;
 return os;
}

int main()
{
  const int size = 3;
  std::array<Student, size> studlist = {Student(123, "Alexander"),
                                        Student(345, "Julius"),
                                        Student(789, "Karl")};

  std::cout << studlist.size() << " Elements in array" << std::endl;

  for (Student itr: studlist){
    std::cout << itr << std::endl;
  }
  return EXIT_SUCCESS;
}
```

```
3 Elements in array
Alexander,123
Julius,345
Karl,789
```

Merke: Vermeiden Sie Raw-Arrays! In der Regel lassen sich die intendierten Strukturen durch die Konstrukte Standardbibliothek wesentlich effizienter realisieren.

Iteratoren

Iteratoren definieren ein allgemeineres Konzept als die zuvor gezeigten Pointer im Umgang mit Arrays. Die darauf anwendbaren Operationen sind:

- Dereferenzierung
- Vergleich `itA == itB`
- Inkrementierung `it++` oder `++it`

wobei die letzten beiden für die "Navigation" über den Daten eines Containers Verwendung findet:

Iteratormethoden	Modus	Unterstützte Container

`vector<int> itr;`
`itr = itr + 2;`
`itr = itr -4;`
`if (itr2 > itr1) ... ;`
`++itr;`
`--itr;`
	Wahlfreier Zugriff / "Random Access Iterator"	vector, deque, array

`list<int> itr;`
`++itr;`
`--itr;`
	Bidirektionaler Iterator	list, set/multiset, map/multimap

`forward_list<int> itr;` `++itr;`	Vorwärts Interator	forward_list
`int x = *itr;`	Eingabe Iterator	
`*itr = 100;`	Ausgabe Iterator	

Anmerkungen:

- Jede Containerklasse integriert eine Implementierung des Iteratorkonzeptes für unbeschränkten Zugriff `iterator` und lesenden Zugriff `const_iterator`. Entsprechend bestehen neben `.begin()` und `.end()` auch `.cbegin()` und `.cend()` (seit C++11).
- Die Pointerarithmetik kann für den wahlfreien Zugriff zum Beispiel mit `itr+=5` umgesetzt werden. Eine übergreifende Lösung bietet die `advance(itr, 5)`-Methode. Analog kann mit `distance(itr1, itr2)` der Abstand zwischen zwei Elementen bestimmt werden.

Ein Haupteinsatzfeld ist die Segmentierung von Containerinhalten nach "innerhalb" und "außerhalb" Elementen, wobei eine bestimmte Funktionalität auf die innerhalb Elemente angewendet wird.

iterators.cpp

```cpp
#include <iostream>
#include <vector>

int main()
{
  std::vector<int> vec {1,2,3,4,5,6,7,8,9};
  vec.erase(vec.begin()+3, vec.end()-1);
  for (std::vector<int>::iterator itr = vec.begin(); itr!=vec.end();
    ){
    std::cout << *itr << ", ";
  }
  return EXIT_SUCCESS;
}
```

```
1, 2, 3, 9,
```

Containerkonzepte

Container sind Objekte, die konzeptionell andere Objekte enthalten. Sie verwenden bestimmte grundlegende Eigenschaften der Objekte (Kopierfähigkeit usw.), hängen aber ansonsten nicht von der Art des Objekts ab, das sie enthalten.

In der STL sind entsprechend alle Container templatisiert. Neben den einzelnen Containerklassen exisitiert ein Set von übergreifenden Algorithmen, die auf die Container angewendet werden können.

```cpp
#include <vector>
#include <deque>
#include <list>
#include <map>      // verantwortlich für set und multiset
#include <set>      //                   map und multimap
#include <iterator>
#include <algorithm>
//...
std::vector<int> a;
std::vector<std::string> b;
```

Welche Arten von Containern werden unterschieden?

1. Sequenzcontainer (*Sequential containers*) ... legen die Daten in einer linearen Ordnung ab. Dabei erfolgt der Zugriff "der Reihe nach" oder anhand einer Indizierung.
2. Associative Container (*Associative containers*) ... verwenden einen beliebigen Datentyp für den Zugriff, der als Schlüssel bezeichnet wird. Dabei wird zwischen ungeordneten und geordneten assoziativen Containern unterschieden.
3. Container Adapter ... passen die in eins und zwei enthaltenen Containerkonstrukte an bestimmte Modelle an, in dem sie die Schnittstellen modifizieren.

Welche Klassifikation sollte in Ihrem Kopf stattfinden?

Darstellung des Entscheidungsprozesses für die Anwendung eines STL-Containers [1]

[1]: Mikael Persson nach einem Entwurf von David Moore Link

Was sind die übergreifenden Methoden der STL Container?

1. Prüfung auf "Leere"

```
if (container.empty()) {std::cout << "Nicht's zu holen!";}
```

2. Zahl der Samples im Container

```
std::cout << container.size();
```

3. Copy constructor

```
vector<int> vec2(vec);
```

4. Löschoperation für alle Einträge

```
container.clear();
```

5. Übertragung der Inhalte zwischen Containern gleichen Typs

```
container_Bs.swap(container_A);
```

Aufgabe: Erklären Sie das vorliegende Verhalten!

initPitfall.cpp

```cpp
#include <vector>
#include <list>
#include <iostream>

int main()
{
  std::list<int> vec (5,2);// {5,2};
  std::cout << vec.size() << std::endl;
  for (auto it: vec){
    std::cout << it << ", ";
  }
  return EXIT_SUCCESS;
}
```

```
5
2, 2, 2, 2, 2,
```

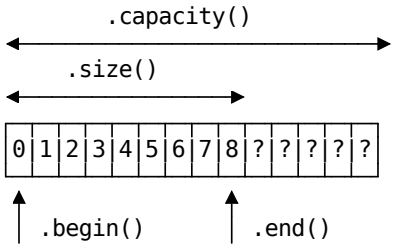
Merke: "() != {}" 🙂

Sequenzcontainer

Container	Idea
`array`	statischer Container
`vector`	Wachstum des zugehörigen Speicher ausschließlich in einer Richtung
`deque`	Wachstum des zugehörigen Speicher beiden Richtungen
`list`	Doppelt verlinkte Einträge
`forward_list`	verlinkte Einträge

Eigenschaft	array	vector	deque	list
dynamische Speichergröße		x	x	x
vorwärts interieren	x	x	x	x
rückwärts iterieren	x	x	x	x
Speicheroverhead	-	ggf.	gering	ja
effizientes Einfügen	-	am Ende	am Ende und am Anfang	überall
Splicen des Containers				X
Iteratoren	wahlfrei	wahlfrei	wahlfrei	bidirektional
Algorithmen	alle	alle	alle	spezifisch

Beispiel `vector`



.

vectorExample.cpp

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

int main()
{
  std::vector<int> vec;  // vec.size() == 0
  vec.push_back(3);
  vec.push_back(1);
  vec.push_back(7);
  //vec.pop_back();

  std::cout << "Vector capacity/size: " << vec.capacity() << " / " <<
    .size() << std::endl;

  // Vector wahlfreier Zugriff
  std::cout << vec[0] << ", " << vec.at(1) << std::endl;
  //              |           |
  // ohne range check    wirft eine Ausnahme bei range Verletzung

  // Wahlfreies Löschen
  // vec.erase(vec.begin()+2);
  // Allgemeinere Lösung (unter Einbeziehung von Listen)
  std::vector<int>::iterator it = vec.begin();
  std::advance(it, 2);
  vec.erase(it);

  // Iteration mittels kompakter Iteratordarstellung (C++11)
  for (auto itr: vec){
      itr +=1;
  }

  for (auto itr = vec.begin(); itr != vec.end(); itr++){
      *itr = *itr + *itr%2;
  }

  std::cout << vec.at(0) << ", " << vec.at(1) << " ..." << std::endl;

  return EXIT_SUCCESS;
}
```

```
Vector capacity/size: 4 / 3
3, 1
4, 2 ...
```

findExample.cpp

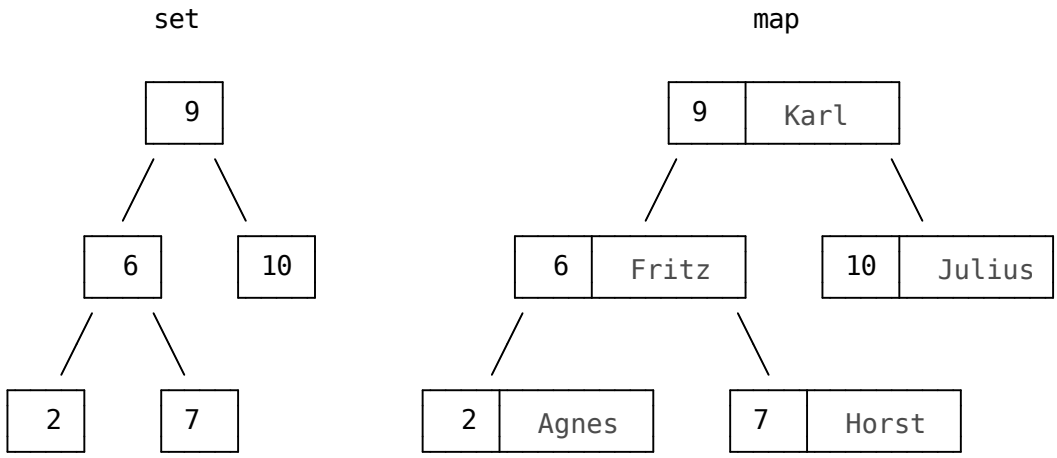
```cpp
#include <vector>
#include <iostream>
#include <algorithm>

int main()
{
  std::vector<int> vec(100000000);
  srand (time(NULL));
  std::generate(vec.begin(), vec.end(), []() {
    return rand() % 100;
  });

  for (std::vector<int>::iterator itr = vec.begin(); itr != vec.end()
    ++itr){
    if (*itr == 99){
      std::cout << "Generic iterator found a 99 " << "at index " <<
        vec.begin() << std::endl;
      break;
    }
  }

  std::vector<int>::iterator it = std::find(vec.begin(), vec.end(), 9
  if (it != vec.end()){
    std::cout << "Element Found" << std::endl;
  } else {
    std::cout << "Element Not Found" << std::endl;
  }

  return EXIT_SUCCESS;
}
```

```
Generic iterator found a 99 at index 57
Element Found
```

Assoziative Container

Container	Idea
`set`	sortierte Menge ohne Dublikate
`map`	eindeutige Schlüssel auf einen Zielwert
`multiset`	sortierte Menge von Elementen, die mehrfach vorkommen
`multimap`	Schlüssel referenzieren ggf. mehrere Einträge

Wie Sie in folgender tabellarischem Vergleich sehen können, realisieren die assoziativen Container ein homogeneres Gesamtbild. Der zentrale Unterschied liegt in der Idee eines Schlüssels, der die Daten selbst repräsentiert oder aber auf den eigentlichen Datensatz verweist.



Eigenschaft	set	map	multiset	multimap
eindeutige Schlüssel	ja		ja	
Schlüssel zu Zielwerten		ja		ja
dynamischer Größe	ja	ja	ja	ja
Overhad pro Element	ja	ja	ja	ja
Speicherlayout	Baum	Baum	Baum	Baum
Iteratoren	wahlfrei	wahlfrei	wahlfrei	wahlfrei
Algorithmen	alle	alle	alle	alle

C++11 erweitert diese Kategorie jeweils unsortierte Darstellungen (vgl. Diagrammdarstellung für die Suche nach geeigneten Containern).
Beispiel `set`

> **Frage:** Warum können die die Elemente nicht manipuliert werden?

setExample.cpp

```cpp
#include <iostream>
#include <algorithm>
#include <set>

int main()
{
  std::set<int> myset;
  myset.insert(3);
  myset.insert(1);
  myset.insert(7);
  myset.insert(7);

  // Evaluate the existence of 7 before insertion
  auto value = 9;
  auto it = myset.find(value);
  if (it == myset.end()){
    std::cout << "No value 9 in current set, added this value" << std
      ;
    myset.insert(value);
    std::cout << "Add " << value << " - ";
  }

  auto ret = myset.insert(value);    // pair<std::set<int>>::iterator
    >
  if (ret.second == false){
    it = ret.first;                // wir haben nun einen Zugriff auf
      Iterator
    std::cout << "Delete " << value << std::endl;
    myset.erase(it);
  }

  for (auto const& itr: myset){
    std::cout << itr << ", ";
  }

  return EXIT_SUCCESS;
}
```

```
No value 9 in current set, added this value
Add 9 - Delete 9
1, 3, 7,
```

Ermitteln Sie alle Einträge, deren Wert größer als 3 und kleiner als 7 ist. Wie würde sich das Vorgehen unterscheiden, wenn wir mit einem `vector`-Container starten würden?

Erfassen Sie die API von `std::upper_bound` und `std::lower_bound` anhand der Dokumentation cppreference.

filterExample.cpp

```cpp
#include <iostream>
#include <algorithm>
#include <set>

int main()
{
  std::set<int> myset {3, 5, 1, 2, 5, 6, 7, 4, 10};
  for (auto const& itr: myset){
    std::cout << itr << ", ";
  }
  std::cout << std::endl;
  std::set<int>::iterator start = std::upper_bound(myset.begin(), mys
    (), 3);  // >
  std::set<int>::iterator end = std::lower_bound(myset.begin(), myset
    (), 7);  // <=
  std::cout << std::distance(start, end) << " Entries found!" << std:
  std::cout << *start << " " << *end << std::endl;
  return EXIT_SUCCESS;
}
```

Anwendungsbeispiel: Führen Sie eine Datenbank der Studierenden einer Universität, die bei korrekter Datenerfassung keine Duplikate zulässt.

setExample.cpp

```cpp
#include <iostream>
#include <algorithm>
#include <set>

class Student{
  private:
    int matrikel;
    std::string name;
  public:
    Student(int matrikel, std::string name): matrikel(matrikel), nam
      ) {};
    friend std::ostream& operator<<(std::ostream& os, const Student&
    bool operator< (const Student &student2) const
    {
      return this->matrikel < student2.matrikel;
    }
};

std::ostream& operator<<(std::ostream& os,  const Student& s){
   os << s.name << "," << s.matrikel;
 return os;
}

int main()
{
  std::set<Student> studlist = {Student(123, "Alexander"),
                                Student(789, "Karl"),
                                Student(345, "Julius"),
                                Student(789, "Karl")};

  for (auto itr: studlist){
    std::cout << itr << std::endl;
  }
  return EXIT_SUCCESS;
}
```

Container-Adapter

Container-Adapter basieren auf Containern, um spezielle Formate zu definieren, die über entsprechende Schnittstellen verfügen. Ein Stack wird beispielsweise mit Hilfe eines Vektors realisiert werden. Der Stack bietet dem Benutzer nur eine eingeschränkte Schnittstelle an. Ein Adapter besitzt keine Iteratoren.

Typ	Beschreibung	Methoden
`std::stack`	Ein Stack speichert seine Elemente in der Reihenfolge ihres Eintreffens und gibt sie in umgekehrter Reihenfolge wieder heraus.	`push(obj)`, `pop()`, `top()`
`std::queue`	FIFO-Speicher	`push(obj)`, `pop()`, `front()`, `back()`
`std::priority_queue`	FIFO-Speicher der den Elementen eine Priorität zuordnet.	`push(obj)`, `pop()`, `front()`, `back()`

setExample.cpp

```cpp
#include <iostream>
#include <stack>
#include <string>

int main()
{

    std::stack<std::string> weekdays;
    weekdays.push("Saturday");
    weekdays.push("Friday");
    weekdays.push("Thursday");
    weekdays.push("Wednesday");
    weekdays.push("Tuesday");
    weekdays.push("Monday");
    weekdays.push("Sunday");

    std::cout<<"Size of the stack: "<<weekdays.size()<<std::endl;

    while(!weekdays.empty()) {
        std::cout<<weekdays.top()<<std::endl;
        weekdays.pop();
    }

   return EXIT_SUCCESS;
}
```

Aufgabe: Tauschen Sie den Container-Adapter Typ aus (`std::queue<string> weekdays`)

Algorithmen

Die Algorithmen, die die STL bereitstellt, lassen sich sowohl auf die vorgestellten Containerkonzepte, wie auch auf native C++ arrays anwenden.

- sortieren
- suchen
- kopieren
- umdrehen
- füllen
- ...

Merke: Die Angabe eines Containerinhaltes mit zwei Iteratoren erfolgt immer halboffen $[begin, end)$. end ist nicht in der Auswahl enthalten.

examplesAlgorithms.cpp

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

int main()
{
  std::vector<int> vec {3, 23, 4, 242, 1, 12, 2, 24};
  auto itr = std::min_element(vec.begin(), vec.end());
  std::sort(vec.begin(), itr);

  std::vector<int> vecNew(3);
  std::copy(itr, vec.end(), vecNew.begin());

  for (auto itr: vecNew){
    std::cout << itr << std::endl;
  }

}
```

Daneben könnnen bei der Interation eigene Funktionen pro Element aufgerufen werden. Dabei unterscheidet man:

- Funktionspointer
- Lambdafunktionen
- Functoren

exampleFunctionCall.cpp

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

bool isOdd(int i){
  return i%2;
}

void add2(int i){
  std::cout << i+2 << ", ";
}

int main()
{
  std::vector<int> vec {6, 23, 4, 242, 1, 12, 2, 23};
  std::find_if(vec.begin(), vec.end(), isOdd);
  std::for_each(vec.begin(), vec.end(), add2);
}
```

```
8, 25, 6, 244, 3, 14, 4, 25,
```

Die Beispiele können durch Lambda-Funktionen analog ausgedrückt werden. Dabei hat man den zusätzlichen Vorteil, dass der zusammengehörige Code nicht aufgesplittet und auf mehrere Positionen verteilt wird. Es sei denn, er wird mehrfach verwendet ...

OOP Konzepte in C++

Versuchen wir uns noch mal an die C#-Welt zu erinnern. Welche Konstrukte gab es innerhalb der Vererbungsstruktur:

Konstrukt	Bedeutung	Wirkung
`abstract class A {}`	Abstrakte Klasse	Der `abstract`-Modifizierer in einer Klassendeklaration definiert, dass die Klasse nur die Basisklasse für andere Klassen sein kann und nicht selbst instanziiert wird. Als abstrakt markierte Member müssen von Klassen, die von nicht abstrakten Klassen abgeleitet wurden, implementiert werden.
`interface IB {}`	Interface	Eine Schnittstelle definiert einen Vertrag. Jede class oder struct, die diesen Vertrag implementiert, muss eine Implementierung der in der Schnittstelle definierten Member bereitstellen. Ab C# 8.0 kann eine Schnittstelle eine Standardimplementierung für Member definieren.
`class A {} : IB, A`	Vererbung	Eine Klasse erbt von abstrakten Klassen, implementierten Klassen oder Interfaces und (re)implementiert deren Member und erweitert den Umfang.

Eine C# Klasse kann nur von einer Basisklasse erben. Gleichzeitig ist aber die Implementierung mehrerer Interfaces möglich.

Vererbung in C++

MinimalExample.cpp

```cpp
#include <iostream>

// Base class
class Shape {
   public:
      Shape(): width(2), height(2) {
        std::cout << "Calling Shape Constructor!" << std::endl;
      }
      Shape(int a, int b): width(a), height(b) {}
      void setWidth(int w) {
         width = w;
      }
      void setHeight(int h) {
         height = h;
      }
   protected:
      int width;
      int height;
};

// Derived class
class Rectangle: public Shape {
   public:
      int getArea() {
         return (width * height);
      }
};

int main(void) {
   Rectangle Rect(5, 6);

   //Rect.setWidth(5);
   //Rect.setHeight(7);
   std::cout << "Total area: " << Rect.getArea() << std::endl;

   return 0;
}
```

```
main.c: In function ‘int main()’:
main.c:30:19: error: could not convert ‘5’ from ‘int’ to ‘Shape’
   30 |    Rectangle Rect(5, 6);
      |                   ^
      |                   |
      |                   int
```

Konstruktoren in der Klassenhierachie

> Die Konstruktoren werden nicht automatisch vererbt!

- Variante 1: Expliziter Aufruf des Konstruktors der Basisklasse: `using Shape::Shape;`
- Variante 2: Individueller Aufruf eines Basisklassenkonstruktors in einem Konstruktor der abgeleiteten Klasse `Rectangle(int width, int height): Shape(width, height)`

Schutzmechanismen für Member

	Member `private`	**Member** `protected`	**Member** `public`
innerhalb der Klasse	X	X	X
abgeleitete Klasse		X	X
außerhalb			X

Und wie verhält es sich bei der Vererbung?

	Member `private`	**Member** `protected`	**Member** `public`
Vererbung `private`	`private`	`private`	`private`
Vererbung `protected`	-	`protected`	`protected`
Vererbung `public`	-	`protected`	`public`

Virtual Funktions

Wie verhält es sich mit dem Methodenaufruf, wenn die spezifischen Implementierungen über verschiedene Elemente verstreut sind?

Polymorphism.cpp

```cpp
#include <iostream>

class A{
  public:
    virtual void print(std::ostream& os) {os << "Base Class" << "\n";
};

class B: public A{
  public:
    void print(std::ostream& os) {os << "Derived Class" <<"\n";}
};

void callPrint(A& object){
  object.print(std::cout);
}

int main(void){
   A a;
   callPrint(a);
    B b;
   callPrint(b);
   return 0;
}
```

```
Base Class
Derived Class
```

`virtual` löst das intuitive Verhalten auf. Anhand der vtables wird dynamisch der Typ einer Funktion zugeordnet. Damit wird zur Laufzeit erkannt, welche Funktion die "richtige" ist.

`overrides` C++11 erleichtert das Programmieren mit virtuellen Methoden

Fehlerfall: - zu unserer mit `override` markierten Methode existiert in der Basisklasse keine Methode - zu unserer mit `override` markierten Methode existiert in der Basisklasse keine virtuelle Methode

Abstrakte Memberfunktionen / Abstrakte Klassen

C++ kennt die Differenzierung zwischen Interfaces und abstrakten Klassen nicht. Eine Klasse wird abstrakt gemacht, indem mindestens eine ihrer Methoden als reine virtuelle Funktion deklariert wird. Eine rein virtuelle Funktion wird spezifiziert, indem `= 0` in ihre Deklaration wie folgt gesetzt wird:

Abstract.cpp

```cpp
#include <iostream>

class Shape {
   public:
      Shape(): width(5), height(6) {}
      virtual double getArea() =  0;
   protected:
      int width;
      int height;
};

class Rectangle: public Shape {
   public:
      double getArea() override {
         return (width * height);
      }
};

int main(void){
   Shape a;
   std::cout <<  a.getArea() << std::endl;
   return 0;
}
```

```
main.c: In function ‘int main()’:
main.c:20:16: error: cannot declare variable ‘a’ to be of abstract type
‘Shape’
   20 |          Shape a;
      |                ^
main.c:3:7: note:   because the following virtual functions are pure
within ‘Shape’:
    3 | class Shape {
      |       ^~~~~
main.c:6:22: note:     ‘virtual double Shape::getArea()’
    6 |       virtual double getArea() =  0;
      |                      ^~~~~~~
```

- Eine Klasse ist abstrakt, wenn sie mindestens eine rein virtuelle Funktion hat.
- von einer abstrakten Klasse können keine Instanzen gebildet werden
- Wenn wir die reine virtuelle Funktion in der abgeleiteten Klasse nicht überschreiben, wird die abgeleitete Klasse auch zur abstrakten Klasse.
- Wir können aber Zeiger und Referenzen vom Typ der abstrakte Klasse haben.

Mehrfachvererbung

Soll eine Klasse von mehreren Basisklassen abgeleitet werden, dann sieht dies in etwa wie folgt aus:

```
class DerivedClass : AccessAttribute1 BaseClass1,
                     AccessAttribute2 BaseClass2,
                     ...
                     AccessAttributen BaseClassn,
{
  /* --- Implementierung weiterer Methoden ---
  ...
  */
};
```

Welche Konsequenzen hat das?

MinimalExample.cpp

```cpp
#include <iostream>

class A
{
public:
  virtual char id(void) {return 'A';}
  virtual int f(void) {return 1;}
  virtual void h(void) {}
};

class B
{
public:
  virtual char id(void) {return 'B';}
  virtual double f(double d) {return d;}
  virtual int i(void) {return 2;}
};

class C : public A, public B
{
};

int main(void)
{
  C c;

  c.id();
  c.f(2.0);
  c.h();        // Individuelle Funktion der Klasse A
  c.i();        // Individuelle Funktion der Klasse B
  return 0;
}
```

```
main.c: In function ‘int main()’:
main.c:27:11: error: request for member ‘id’ is ambiguous
   27 |         c.id();
      |           ^~
main.c:14:22: note: candidates are: ‘virtual char B::id()’
   14 |         virtual char id(void) {return 'B';}
      |                      ^~
main.c:6:22: note:                 ‘virtual char A::id()’
    6 |         virtual char id(void) {return 'A';}
      |                      ^~
main.c:28:11: error: request for member ‘f’ is ambiguous
   28 |         c.f(2.0);
      |           ^
main.c:15:24: note: candidates are: ‘virtual double B::f(double)’
   15 |         virtual double f(double d) {return d;}
      |                        ^
main.c:7:21: note:                 ‘virtual int A::f()’
    7 |         virtual int f(void) {return 1;}
      |                     ^
```

Funktionsaufruf	Resultat	Bedeutung
`c.id();`	`error: request for member ‘id’ is ambiguous`	
`c.f(2.0);`	`error: request for member ‘f’ is ambiguous`	Trotz unterschiedlicher Signaturen kann der Compiler die "passende" Funktion aus der Klasse B nicht zuordnen.

Bei jedem Aufruf lässt sich aber explizit angeben, welche Funktion von welcher Basisklasse gewünscht ist. Der Bereichsauflösungsoperator `::` unter Benennung der Klasse gibt die konkrete Funktion an.

```
int main(void)
{
  C c;

  c.A::id();
  c.B::f(2.0);
  c.h();
  c.i();
  return 0;
```

```
  return 0;
}
```

Diese Aufgabe kann aber natürlich auch in eine überladene Memberfunktion verlagert werden:

```
class C : public A, public B
{
  public:
    //A::f
    int f(void) {return A::f();}

    //B::f
    double f(double d) {return B::f(d);}

    //B::g
    using B::g;

    //A::id
    using A::id;
};
```

Aufgabe der Woche

1. Verschaffen Sie sich einen Überblick zu den Lambdafunktionen. Ermitteln Sie die Syntax und die Anwendung. Implementieren Sie ein Beispiel, dass unsere Studentencontainer nach spezifischen Merkmalskombinationen durchsucht.
2. Recherchieren Sie in Arbeiten zur Performance der verschiedenen Containerkonstrukte. Welchen Einfluss haben verschiedene Datenformate und Operationen auf die Größe des notwendigen Speichers und die Dauer der Ausführung.