Templates

❑ Motivation

❑ Generische Parameter

❑ Template-Funktionen

❑ Template-Klassen

Templates - Motivation

❑ Normale Vorgehensweise

- An mehreren Stellen benutzte Programmabschnitte in Unterprogramm zusammenfassen

❑ Problem:

- Was ist bei gleichen Abschnitten mit unterschiedlichen Typen?

❑ Beispiel:

```
int     sqr (int i)     {return i*i;}
double  sqr (double d)  {return d*d;}
```

❑ Bisherige Lösung: Makros

```
#define sqr(X)  ((X)*(X))
```


Templates - Generische Parameter

❑ Nötig:

- Parameter, deren Typ bei der Deklaration und Implementierung noch nicht bekannt ist

  $\Longrightarrow$ Generische Parameter

❑ Wünschenswert:

- Typprüfung trotzdem möglich
- Kein erheblicher Effizienzverlust

Template-Funktionen

❑ In C++: Template-Funktionen

```
template <class T> T sqr (T i)
{
  return i*i;
}

void main ()
{
  int z = 3;
  int z1 = sqr (++z);  // ok
  Complex c(2,5);
  c = sqr (c);         // ok, wenn Complex operator* hat
  char* s = "";
  s = sqr (s);         // Fehler !!
}
```

Template-Funktionen

```
template <class T> T* Duplicate (T* object)
{
   T* retVal = new T;
   (*retVal) = (*object);
   return retVal;
}
char* Duplicate (char* object)
{
   char* retVal = new char[strlen(object)+1];
   strcpy (retVal, Object);
   return retVal;
}
void main ()
{
   char* x = Duplicate (“Hallo“);
   Temperatur t;
   Temperatur t2 = Duplicate (&t);
   delete x; delete t2;
}
```


Template-Funktionen

Template-Funktionen in C++ sind:

- Generische Funktionen mit strenger Typprüfung
- Überladbar
- Eigene Funktionsinstanz für jeden verwendeten Datentyp
  - Der Quelltext taucht nur einmal auf, die Übersetzung erfolgt mehrfach
  - Der Implementationsquelltext muß in jedem Modul vorliegen

    ⇒ Implementation in der .h-Datei
- Geeignet zur Wiederverwendung von Verfahrensweisen

Klassen-Templates: Motivation

❏ Viele Strukturen werden oft für verschiedene Typen benötigt

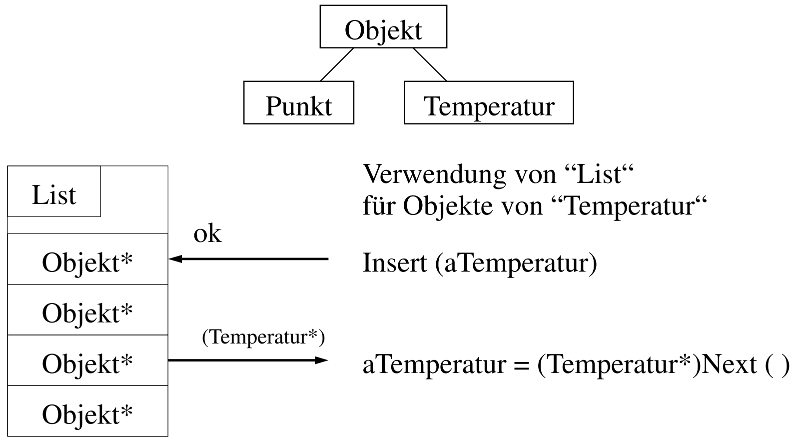
❏ Beispiele:

- Verkette Liste
- Sortierte Liste

❏ Bisherige Möglichkeiten der Realisierung:

- Einzelne Programmierung
- Verwaltung mit void* - Pointern
- Verwaltung von Oberklassen - Pointern

Klassen-Templates: Motivation



Template-Klassen

❏ Template-Klassen sind Klassen, die

- Komponenten variablen Typs haben oder
- von variablen Klassen abgeleitet sind

❏ Vorteile

- Genau so schnell wie Einzelimplementationen
- Typprüfung immer aktiv
- Ideal für Klassenbibliotheken geeignet

Template-Klassen

```
template <class T, int size> class Stack
{
   T feld[size];
   T* p;
public:
   Stack ()        { p=feld; }
   void Push (T a) { *p++ = a; }
   T Pop ();
};

template <class T, int size>
T Stack<T, size>::Pop ()
{
   return *--p;
}
```


Template-Klassen

```
void main ()
{
   Stack <int, 30> myStack;
   myStack.Push (2);
   myStack.Push (3);
   int x = myStack.Pop ();   // Wert 3
   int y = myStack.Pop ();   // Wert 2
}
```


Template-Klassen

❏ Implementierbar:

- Listen jeder Art:

```
template <class T> class List
{
public:
   T* First ();
   T* Next ();
   void Insert (T*);
};
void main ()
{
   List<Temperatur> list;
   for (Temperatur* t=list.First(); t; t=list.Next())
        // Tue irgendetwas mit t
}
```

Template-Klassen

❑ Erweiterung eines Templates

```
template <class T> class SortableList : public List<T>
{
   ...
   void Sort ();
};
```

❑ Konkretisierung eines Templates

```
class MyIntegerList : public List<int>
{
   ...
};
```


Templates - Zusammenfassung

❑ Vorteile:

- Strenge Typprüfung
- Praktisch keine Doppelimplementationen mehr nötig

❑ Template - Klassen sind geeignet für

- Grundlegende Verwaltungsmechanismen
- Klassenbibliotheken

lokale Klassen

❑ Klassen und andere Strukturen, die im wesentlichen von einer Klasse gebraucht werden, kann man lokal definieren:

```
class Ruderboot: public Boot {
public:
   class Ruder
   {      ...
          Ruder();
   };
   void DoSomething() {
          Ruder x;  .....
   }
};
Ruderboot::Ruder::Ruder() {....}

main() {
   Ruder x;// Fehler: gibts nicht
   Ruderboot::Ruder x;// ok
}
```


abstrakte Klassen

❑ Abstrakte Klassen:

- nicht alle Methoden sind implementiert
- die Klasse ist daher syntaktisch nicht instantiierbar
- Unterklassen sind erst instantiierbar, wenn alle deklarierten Methoden implementiert wurden.

❑ Abstrakte Klassen dienen

- als nicht instantiierbare Oberklasse
- als Definition einer Typschnittstelle

abstrakte Klassen

```
class Boot
{
public:
     virtual double Hoechstgeschwindigkeit() = 0;   // Nicht implementiert
     double Normalgeschwindigkeit() {  return Hoechstgeschwindigkeit() / 2;  }
};

class Motorboot: public Boot
{
     virtual double Hoechstgeschwindigkeit()  {  return 50;  }
};

main()
{
     Boot b;                                  // verboten
     Motorboot mb;                            // ok
     Boot *bp = &mb;                          // ok
     cout << bp->Hoechstgeschwindigkeit();    // ok
}
```


Mehrfachvererbung

- ❏ Einführung
- ❏ Vererbungsbeispiele
- ❏ Mehrfachvererbung in C++
- ❏ Virtuelle Basisklassen
- ❏ Namenssuchstrategien bei Mehrdeutigkeiten
- ❏ Lokale Klassen
- ❏ Abstrakte Klassen

Mehrfachvererbung

- ❏ Vererbung bedeutet "is a" (ist ein Spezialfall von)
- ❏ Einfachvererbung:
  - Jede Klasse ist von maximal einer Basisklasse abgeleitet
- ❏ Mehrfachvererbung:
  - Eine Klasse kann von mehreren Klassen abgeleitet sein
  - Mehrdeutigkeiten bei Methodenaufrufen

Mehrfachvererbung - Einführung

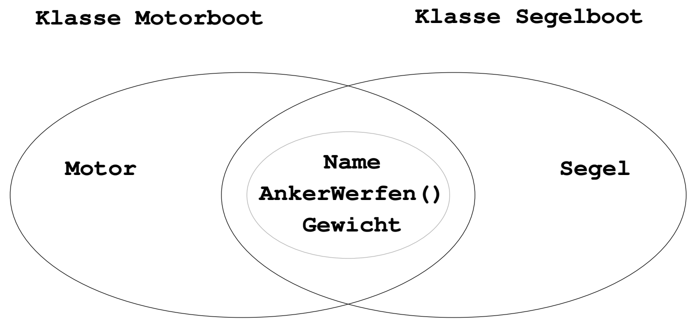
- ❏ Eine erbende Klasse enthält jede ihrer Oberklassen als impliziten Bestandteil
- ❏ Der Namensraum der Oberklasse(n) wird mit eingeblendet
- ❏ Virtuelle Methodenaufrufe werden an die "echte" Klasse zugestellt
- ❏ Bei der Konstruktion von Objekten kann ein Konstruktoraufruf für Oberklassen angegeben werden
- ❏ Bei der Destruktion von Objekten wird automatisch der Destruktor der Oberklassen aufgerufen

Mehrfachvererbung - Einführung

```
class Hardwareport {
   int portNumber;
public:
   Hardwareport (int port): portNumber(port) {...}
   virtual ~Hardwareport (){ }
   virtual char Encode(char c){return c;}
   virtual void Put(char c) { .... out(portNumber,Encode(c));... }
};
class CodedHardwareport: public Hardwareport {
   Key myKey;
public:
   CodedHardwareport(int port, Key k) : myKey(k), Hardwareport (port) { }
   virtual char Encode(char c){return code(c,myKey);}
};
main() {
   Hardwareport* myPort=new CodedHardwareport(aKey,2345);
   myPort->Put('#');
   delete myPort;
}
```

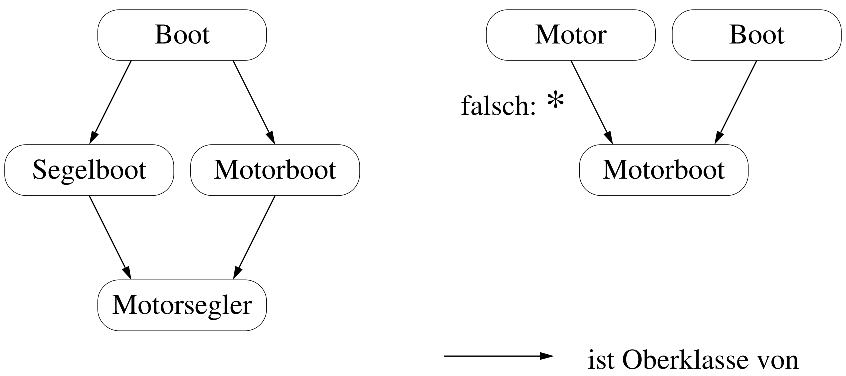

Mehrfachvererbung - Oberklasse

❑ Eine gemeinsame Oberklasse von zwei Klassen sollte die Schnittmenge der Attribute enthalten



Mehrfachvererbung

RICHTIG FALSCH



Mehrfachvererbung

```
class Boot
{
      protected:   String name;
      public:      Boot(String n):  name(n)  { }
};
```

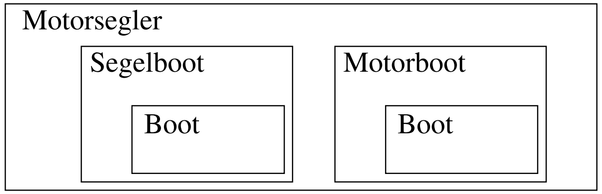
```
class Segelboot: public Boot
{
protected:
      Segel meinSegel;
public:
      Segelboot(String n): Boot(n) { }
};
```

```
class Motorboot: public Boot
{
protected:
      Motor meinMotor;
public:
      Motorboot(String n): Boot(n) { }
};
```

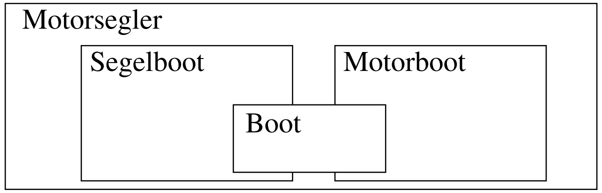
```
class Motorsegler: public Segelboot, public Motorboot
{
public:
      Motorsegler(String n): Segelboot(n), Motorboot(n) { }
};
```

Mehrfachvererbung - virtuelle Basisklassen

❏ Implementation mit „normalen“ Vererbungsregeln



❏ Implementation mit virtueller Basisklase



Mehrfachvererbung - virtuelle Basisklassen

```
class Boot
{
      protected:   String name;
      public:      Boot(String n):  name(n)  { }
};
```

```
class Segelboot: virtual public Boot
{
protected:
     Segel meinSegel;
public:
     Segelboot(String n): Boot(n) { }
};
```

```
class Motorboot: virtual public Boot
{
protected:
     Motor meinMotor;
public:
     Motorboot(String n): Boot(n) { }
};
```

```
class Motorsegler: public Segelboot, public Motorboot
{
public:
    Motorsegler(String n): Boot(n),Segelboot(n),Motorboot(n) { }
};
```


Mehrfachvererbung - Mehrdeutigkeiten

```
class Boot {
     public:      virtual void PersonBetrittBoot( ) = 0;
};
class Segelboot: virtual public Boot {
     public:      virtual void PersonBetrittBoot( ) {........}
};
class Motorboot: virtual public Boot {
     public:      virtual void PersonBetrittBoot( ) {........}
};
class Motorsegler: public Segelboot, public Motorboot  {
     public:      Motorsegler(String n): Boot(n), Segelboot(n), Motorboot(n) { }
};
void main() {
     Motorsegler m;
     Boot* b=&m;
     m.PersonBetrittBoot();                 // verboten, mehrdeutig
     m.Motorboot::PersonBetrittBoot();      // ok
     b->PersonBetrittBoot();                // erlaubt, aber mehrdeutig!
}
```