4 Generische Programmierung

4.1 Klassen-Templates (*)
4.2 Funktions-Templates (*)
4.3 Besondere Programmiertechniken (Smart Pointer)

(*) Auch in dieser Datei.

Intelligente Zeiger

```
bool checkManyThings() {
   Student* p_s1 = new Student;
   Student  s2;
   …
   if (….)   return false;

   DoVeryComplicatedThings(s2);
   try {
       DoOtherVeryComplicatedThings(* p_s1);
        }
   catch (exception& e) {
        cout << "Das war wohl nix;";
        throw e;
    }
    …
    delete p_s1;
    return true;
   }
```

Was könnte hier alles schief gehen?

Warum haben wir „unsauber“ programmiert?

Intelligente Zeiger

Man muss am Ende der Funktion oder auch danach selber dafür sorgen, die nötigen Aufräumarbeiten durchzuführen.
Bei normalen Objekten (keine Zeiger) übernimmt das der Compiler durch Aufruf des richtigen Destruktors.

Bei Zeigern ist dieses nicht nur umständlich, sondern auch sehr gefährlich, weil beim Auftreten einer Ausnahme die Funktion sogar mittendrin verlassen werden kann. Das Aufräumen wird dann nämlich nicht durchgeführt (es sei denn, man fängt die Ausnahme in der Funktion und führt wiederum selbst die Aufräumarbeiten durch).

Hier helfen nun Smart-Pointer, sogenannte intelligente Zeiger, die, wenn sie zerstört werden, auch das Objekt zerstören, auf das sie verweisen.

Einführung

```
template<class T>
class SmartPtr {
    T* ptr;
  public:
    SmartPtr  (T* p=nullptr) : ptr(p) {}
    ~SmartPtr () { delete ptr; }
};

void f() {
  SmartPtr<double> px = new double; // Aufruf des Konstruktors
  . . .
}                                   // Aufruf des Destruktors
```

Smart Pointer sind selbst definierte Zeiger, die wie "`normale"' Zeiger verwendet werden können, aber mit zusätzlichen Fähigkeiten ausgestattet sind:

Smart-Pointer (2)

Um mit solchen Smart-Pointern ähnlich wie mit normalen Zeigern arbeiten zu können, müssen natürlich weitere Funktionen und Operatoren zur Verfügung gestellt werden, d.h. zumindest:

- Kopierkonstruktor,
- Zuweisungsoperator,
- Dereferenzierungsoperator und
- der Operator für den Komponentenzugriff.

Der folgende Text zeigt eine entsprechende Klasse und eine kleine Anwendung, die die Verwendung demonstriert:

Smart Pointer (3)

```
template<class T>
class SmartPtr {
    T* ptr;
  public:
    SmartPtr(T* p=0) : ptr(p) {}
    SmartPtr(SmartPtr& ap)    { ptr = ap.ptr;         ap.ptr = nullptr; }
    ~SmartPtr()                  { delete ptr; }

    SmartPtr<T>& operator= (SmartPtr& ap) { // Achtung das gibt es
                             nicht bei unique_ptr (siehe später) dort bei &&
      if (this != &ap) {
        delete ptr;       ptr = ap.ptr;          ap.ptr = nullptr;
      }
      return *this;
    }
    T& operator*  () { return *ptr; }
    T* operator-> () { return ptr; }
};
```

Smart Pointer (3)

```
template<class T>
class SmartPtr {
    T* ptr;
  public:
    SmartPtr(T* p=0) : ptr(p) {}
    SmartPtr(SmartPtr& ap)    { ptr = ap.ptr;         ap.ptr = nullptr; }
    ~SmartPtr()                  { delete ptr; }
    SmartPtr<T>& operator= (SmartPtr& ap) = delete;
    SmartPtr<T>& operator= (SmartPtr&& ap) {if (this != &ap) {
        delete ptr;       ptr = ap.ptr;          ap.ptr = nullptr;
      }
      return *this;
    }
    T& operator*  () { return *ptr; }
    T* operator-> () { return ptr; }
};
```

Smart Pointer (4)

```
struct X {
  int x;
  X()  { cout << "+A "; }
  ~X() { cout << "-A "; }
};
void f() {
  X *pa          = new X;
  SmartPtr<X> pb = new X; SmartPtr<X>  pc;
  pa->x = 123; cout << pa->x << " ";
  pb->x = 321; cout << pb->x << " ";
  pc = pb;     cout << pc->x << " ";
}

int main() { f(); return 0;}
```

Die Ausgabe des Programms ist:

+A +A 123 321 321 -A

STL-Klasse unique_ptr // früher auto_ptr (vor C11)

unique-Zeiger sind die standardisierte Form (aus der STL) der intelligenten Zeiger (smart pointer).
Der folgende Code zeigt einen Ausschnitt aus der entsprechenden Klassenschnittstelle:

```
template<class X> class unique_ptr {
  public:
    typedef X element_type;

    // Konstruktoren und Destruktor:
    explicit unique_ptr(X* p =0);
    ~unique_ptr();
    template<class Y> unique_ptr(const unique_ptr<Y>&);
```

STL-Klasse unique_ptr (2)

```
// Dereferenzierung:
    X& operator*() const;
    X* operator->() const;

    // Zugriff auf Freigabe:
    X* get() const;
    X* release() const;
};
```

Am Ende der Funktion f wird der Destruktor des Unique-Zeiger-Objektes ptr aufgerufen, das sozusagen der Besitzer des Objekte vom Typ SomeWhat ist und bei seiner Destruktion dann explizit das SomeWhat-Objekt freigibt.

Grundgedanke und Verwendung:

```
#include <memory>
using namespace std;

void f() {
  unique_ptr<SomeWhat> ptr = new SomeWhat;
  . . .
};  // hier wird das Objekt wieder "automatisch" freigegeben
```

Intelligente Zeiger (2)

```
unique_ptr<Stack<int> > p_st0(new Stack<int>);
```

Die Zerstörung des mit new erzeugten Stacks erfolgt sobald der Destruktor von p_st0 aufgerufen wird.

Der Zugriff mit p_st0 ist wie bei normalen Zeigern möglich, da die Operatoren * und -> überladen wurden.

```
p_st0-> Empty()
(*p_st0). Push(20);
```

Es gibt keine zwei unique_ptr, die auf das gleiche Objekt verweisen, da es sonst zwei „Zerstörer“ geben würde.

Bei Kopieren eines unique_ptr geht das verwaltete Objekt daher in den Besitz der Kopie über.

Clicker: unique_ptr-Nutzung

```
class Z {
public:
   Z(int id=9): m_id(id) { cout <<"+Z" << id << " ";}
   ~Z()  { cout <<"-Z" << m_id << " "; }
   Z(const Z& z2):m_id(z2.m_id) {
      cout <<"+ZC" << m_id << " ";  }
   void use() const{ cout << "use:" << m_id << " "; }
private:
   int m_id;
};
void SimpleTest() {
   unique_ptr<Z> up(new Z(11));
   Z* pz = new Z(4);
}
```

Ausgabe?

1. +Z11 +Z4 use:4 use:11 -Z11
2. +Z11 +Z4 -Z11
3. +Z11 -Z11
4. +Z11 +Z4

Clicker: unique_ptr-Nutzung

```
class Z {
public:
   Z(int id=9): m_id(id) { cout <<"+Z" << id << " ";}
   ~Z()  { cout <<"-Z" << m_id << " "; }
   Z(const Z& z2):m_id(z2.m_id) {
      cout <<"+ZC" << m_id << " ";  }
   void use() const{ cout << "use:" << m_id << " "; }
private:
   int m_id;
};
void SimpleTest() {
   unique_ptr<Z> up(new Z(11));
   Z* pz = new Z(4);
}
```

Ausgabe?

1. +Z11 +Z4 use:4 use:11 -Z11
2. +Z11 +Z4 -Z11
3. +Z11 -Z11
4. +Z11 +Z4

+Z11 +Z4 use:4 use:11 -Z11// +Z11 +Z4 -Z11// +Z11 -Z11 // +Z11 +Z4

Clicker: Smart-Pointer Kopieren (4)

```
void f1(unique_ptr<Z>& ap){
   ap->use();
}
void CopyTestAuto2() {
   unique_ptr<Z> up(new Z(11));
   up->use();
   f1(up);
   // up->use(); // kein Problem
   cout << "Ende ";
}
```

Ausgabe?

1. +Z11 use:11 use:11 Ende -Z11
2. +Z11 use:11 use:11 Ende
3. +Z11 use:11 use:11 -Z11 Ende
4. +Z11 Ende

Siehe Datei WS2020_Ueb1.cpp

Clicker: Smart-Pointer Kopieren (4)

```
void f1(unique_ptr<Z>& ap){
   ap->use();
}
void CopyTestAuto2() {
   unique_ptr<Z> up(new Z(11));
   up->use();
   f1(up);
   // up->use(); // kein Problem
   cout << "Ende ";
}
```

Ausgabe?

1. +Z11 use:11 use:11 Ende -Z11
2. +Z11 use:11 use:11 Ende
3. +Z11 use:11 use:11 -Z11 Ende
4. +Z11 Ende

+Z11 use:11 use:11 Ende -Z11 // +Z11 use:11 use:11 Ende
// +Z11 use:11 use:11 -Z11 Ende// +Z11 Ende

Clicker: Smart-Pointer Kopieren (5)

```
void f3(unique_ptr<Z> ap){  // als Wert
   ap->use();
}
void CopyTestAuto4() {
   unique_ptr<Z> up(new Z(11));
   up->use();
   f3(up);
   cout << "Ende ";
}
```

Ausgabe?

1. +Z11 use:11 use:11 Ende -Z11
2. +Z11 use:11 use:11 Ende
3. Compilerfehler
4. +Z11 Ende

Siehe Datei WS2020_Ueb2.cpp

Clicker: Smart-Pointer Kopieren (5)

```
void f3(unique_ptr<Z> ap){  // als Wert
   ap->use();
}
void CopyTestAuto4() {
   unique_ptr<Z> up(new Z(11));
   up->use();
   f3(up);
   cout << "Ende ";
}
```

Ausgabe?

1. +Z11 use:11 use:11 Ende -Z11
2. +Z11 use:11 use:11 Ende
3. Compilerfehler
4. +Z11 Ende

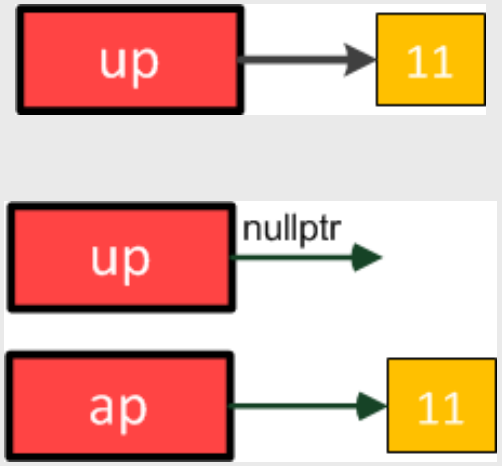
Hier bekommen wir einen Compilerfehler.
unique_ptr haben keinen Kopier-Konstruktor und keinen Zuweisungsoperator.

Clicker: unique_ptr haben Verschiebe-Konstruktor

```
void f5(unique_ptr<Z> ap){
   ap->use();
}
void CopyTestAuto5() {
   unique_ptr<Z> up(new Z(11));
   up->use();

   // f5(up); // Compilerfehler

   f5(  move(up)  );   // wandelt in R-Value (temp. Variable um)
    //f5(  ::move(up)  ); // macht das gleiche
   cout << "Ende ";
}
```



//up->use(); / wäre nun ein Fehler, da up ungültig

Siehe Code in Uebung4UniqueMove.cpp

Clicker: unique_ptr haben Verschiebe-Konstruktor (2)

```
void f6(unique_ptr<Z> ap) // als Wert
{
   ap = unique_ptr<Z>(new Z(4));
   cout << "f6 ";
}
void CopyTestAuto6() {
   unique_ptr<Z> up(new Z(12));
   f6(move(up));
   cout << "Ende ";
}
```

Ausgabe?

1. +Z12 +Z4 -Z12 f6 Ende -Z4
2. +Z12 +Z4 f6 -Z12 -Z4 Ende
3. +Z12 +Z4 -Z12 f6 -Z4 Ende
4. +Z12 +Z4 -Z12 -Z4 f6 Ende
5. +Z12 +Z4 f6 -Z4 Ende

Clicker: unique_ptr haben Verschiebe-Konstruktor (2)

```
void f6(unique_ptr<Z> ap) {// als Wert {
   ap = unique_ptr<Z>(new Z(4));
   cout << "f6 ";
}
void CopyTestAuto6() {
   unique_ptr<Z> up(new Z(12));
   f6(move(up));
   cout << "Ende ";
}
```

Ausgabe?

1. +Z12 +Z4 -Z12 f6 Ende -Z4
2. +Z12 +Z4 f6 -Z12 -Z4 Ende
3. +Z12 +Z4 -Z12 f6 -Z4 Ende
4. +Z12 +Z4 -Z12 -Z4 f6 Ende
5. +Z12 +Z4 f6 -Z4 Ende

1. +Z12 +Z4 -Z12 f6 Ende -Z4
2. +Z12 +Z4 f6 -Z12 -Z4 Ende
3. +Z12 +Z4 -Z12 f6 -Z4 Ende // in ap = .. Wird temp Objekt zugewiesen also operator==(&&) wird aufgerufen, siehe nächsten Folien
4. +Z12 +Z4 -Z12 -Z4 f6 Ende
5. +Z12 +Z4 f6 -Z4 Ende

Move-Semantik

```
 template<typename T>
 class unique_ptr {
   unique_ptr(const unique_ptr&) = delete;  // Aufruf verboten
   unique_ptr(unique_ptr&&);

};
```

move gibt es in der STL in der Datei <memory>

```
 template<typename T>
T&& move(T& tt) {return tt;}
```

Clicker: unique_ptr haben Verschiebe-Zuweisung

```
void f7(unique_ptr<Z> aup1){
   unique_ptr<Z> up2(new Z(4));
   aup1 = up2;
   cout << "f7 ";
}
void CopyTestAuto7() {
   unique_ptr<Z> up(new Z(12));
   f7(::move(up));
   cout << "Ende ";
}
```

Ausgabe?

1. Compilerzeitfehler
2. +Z12 f7 +Z4 -Z12 -Z4 Ende
3. +Z12 +Z4 -Z12 f7 -Z4 Ende
4. -Z12 -Z4 f7 Ende
5. Laufzeitfehler

Clicker: unique_ptr haben Verschiebe-Zuweisung

```
void f7(unique_ptr<Z> aup1){
   unique_ptr<Z> up2(new Z(4));
   aup1 = up2;
   cout << "f7 ";
}
void CopyTestAuto7() {
   unique_ptr<Z> up(new Z(12));
   f7(::move(up));
   cout << "Ende ";
}
```

Ausgabe?

1. Compilerzeitfehler
2. +Z12 f7 +Z4 -Z12 -Z4 Ende
3. +Z12 +Z4 -Z12 f7 -Z4 Ende
4. -Z12 -Z4 f7 Ende
5. Laufzeitfehler

1. Compilerfehler; Zuweisung bei unique_ptr verboten
2. +Z12 f7 +Z4 -Z12 -Z4 Ende
3. +Z12 +Z4 -Z12 f7 -Z4 Ende
4. -Z12 -Z4 f7 Ende
5. Laufzeitfehler

Intelligente Zeiger: Kopieren (2)

Entsprechendes wie für den Verschiebe-Konstruktor gilt für den Zuweisungsoperator als Verschiebe-Zuweisng operator=

```
p_st2 = p_st1;
   // STL von VS 2013
_Myt& operator=(_Myt&& _Right)
{// assign by moving _Right
if (this != &_Right)
   {// different, do the move
   reset(_Right.release());
   this->get_deleter() =
         _STD forward<_Dx>(_Right.get_deleter());
    }
return (*this);
}
```

Besitzt p_st2 bereits ein Objekt, wird dieses mit delete zuvor zerstört.

Clicker: unique_ptr haben Verschiebe-Zuweisung (2)

```
void f7(unique_ptr<Z> aup1){
   unique_ptr<Z> up2(new Z(4));
   aup1 = move(up2);
   cout << "f7 ";
}
void CopyTestAuto7() {
   unique_ptr<Z> up2(new Z(12));
   f7(::move(up2));
   cout << "Ende ";
}
```

Ausgabe?

1. Compilerzeitfehler
2. +Z12 f7 +Z4 -Z12 -Z4 Ende
3. +Z12 +Z4 -Z12 f7 -Z4 Ende
4. -Z12 -Z4 f7 Ende
5. Laufzeitfehler

Siehe Code in Uebung5UniqueMoveAss.cpp

Clicker: unique_ptr haben Verschiebe-Zuweisung (2)

```
void f7(unique_ptr<Z> aup1){
   unique_ptr<Z> up2(new Z(4));
   aup1 = move(up2);
   cout << "f7 ";
}
void CopyTestAuto7() {
   unique_ptr<Z> up2(new Z(12));
   f7(move(up2));
   cout << "Ende ";
}
```

Ausgabe?

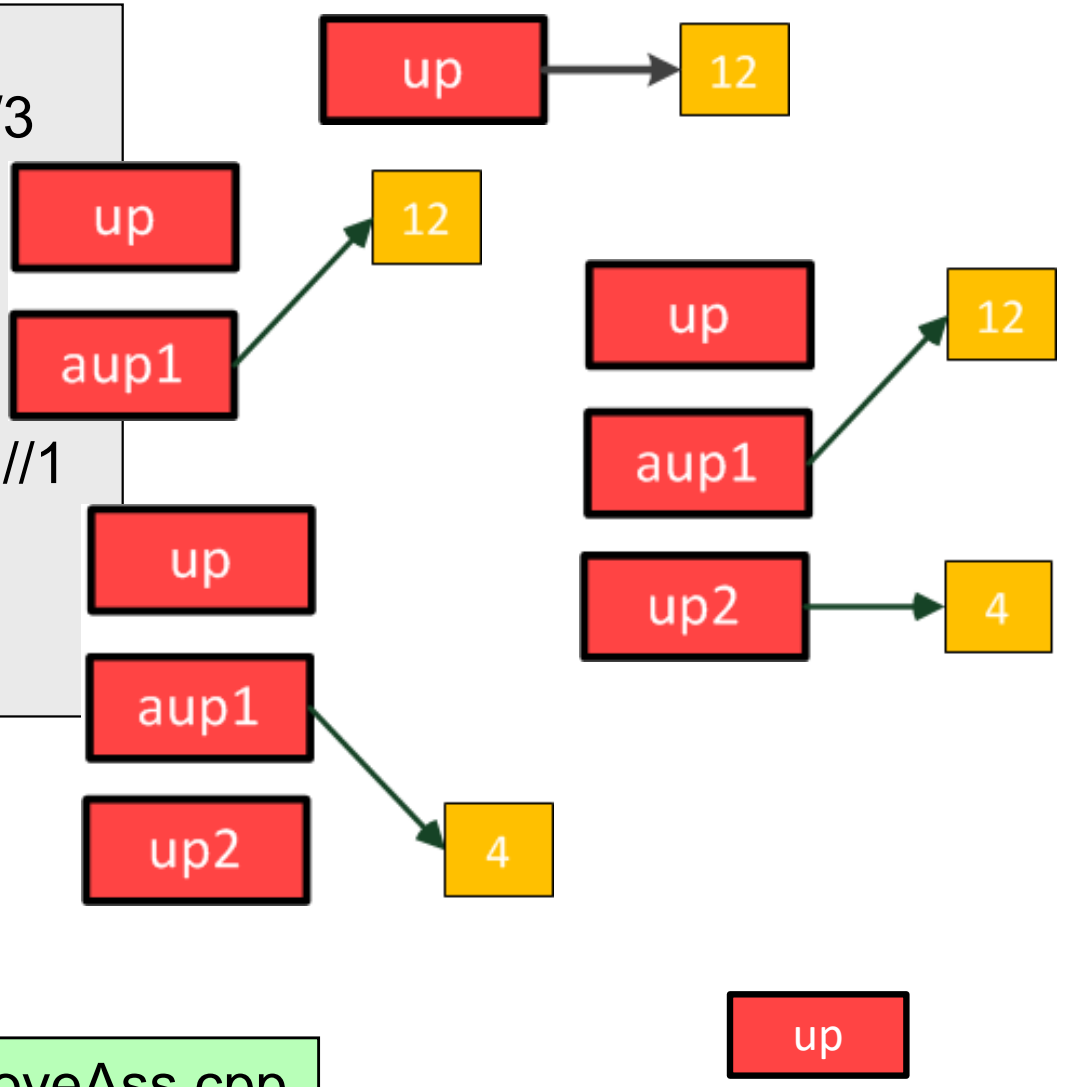
1. Compilerzeitfehler
2. +Z12 f7 +Z4 -Z12 -Z4 Ende
3. +Z12 +Z4 -Z12 f7 -Z4 Ende
4. -Z12 -Z4 f7 Ende
5. Laufzeitfehler

1. Compilerfehler; Zuweisung bei unique_ptr verboten
2. +Z12 f7 +Z4 -Z12 -Z4 Ende
3. **+Z12 +Z4 -Z12 f7 -Z4 Ende**
4. -Z12 -Z4 f7 Ende
5. Laufzeitfehler

Siehe Code in Uebung5UniqueMoveAss.cpp

Clicker: unique_ptr haben Verschiebe-Zuweisung (2)

```
void f7(unique_ptr<Z> aup1){ //2
   unique_ptr<Z> up2(new Z(4)); //3
   aup1 = move(up2); //4
   cout << "f7 ";
} /5
void CopyTestAuto7() {
   unique_ptr<Z> up1(new Z(12)); //1
   f7(::move(up1));
   cout << "Ende ";
}
```



Siehe Code in Uebung5UniqueMoveAss.cpp

Clicker: unique_ptr mit Methode get

```
class Z {
…
void use() const{ cout << "use:"
                  << m_id << " "; }
..
};
void WS2020_Ueb3 () {
   unique_ptr<Z> up(new Z(12));
   Z* p = up.get(); // liefert den internen Zeiger zurück
   p->use();
   cout << "Ende ";
   up->use();
}
```

Ausgabe?

1. +Z12 use:12 Ende use:12 -Z12
2. +Z12 use:12 Ende use:12
3. +Z12 -Z12 use:12 Ende use:12
4. Compilerfehler
5. Laufzeitfehler

Siehe Datei WS2020_Ueb3.cpp

Clicker: unique_ptr mit Methode get

```
class Z {
…
void use() const{ cout << "use:"
                  << m_id << " "; }
..
};
void CopyTestAuto8() {
   unique_ptr<Z> up(new Z(12));
   Z* p = up.get(); // liefert den
          // internen Zeiger zurück
   p->use();
   cout << "Ende ";
   up->use();
}
```

Ausgabe?

1. +Z12 use:12 Ende use:12 -Z12
2. +Z12 use:12 Ende use:12
3. +Z12 -Z12 use:12 Ende use:12
4. Compilerfehler
5. Laufzeitfehler

1. +Z12 use:12 Ende use:12 -Z12
2. +Z12 use:12 Ende use:12
3. +Z12 -Z12 use:12 Ende use:12
4. Compilerfehler
5. Laufzeitfehler

Clicker: unique_ptr mit Methode release

```
void CopyTestAuto9() {
   unique_ptr<Z> up(new Z(12));
   Z* p = up.get();
   up.release(); // setzt internen Zeiger auf nullptr; ruft aber NICHT delete auf
   p->use();
   up->use();
   cout << "Ende ";
}
```

```
class Z {
…
void use() const{
cout << "use:"  << m_id << " "; }
..
};
```

Ausgabe?

1. +Z12 use:12 use:12 Ende -Z12
2. +Z12 Bumm
3. +Z12 use:12 Bumm
4. +Z12 use:12 use:12 Ende Bumm -Z12
5. Compilerfehler

Siehe Datei WS2020_Ueb4.cpp

Clicker: unique_ptr mit Methode release

```
void CopyTestAuto9() {
   unique_ptr<Z> up(new Z(12));
   Z* p = up.get();
   up.release(); // setzt internen Zeiger auf nullptr; ruft aber NICHT delete auf
   p->use();
   up->use();
   cout << "Ende ";
}
```

Ausgabe?

1. +Z12 use:12 use:12 Ende -Z12
2. +Z12 Bumm
3. +Z12 use:12 Bumm
4. +Z12 use:12 use:12 Ende Bumm -Z12
5. Compilerfehler

1. +Z12 use:12 use:12 Ende -Z12
2. +Z12 Bumm
3. **+Z12 use:12 Bumm**
4. +Z12 use:12 use:12 Ende Bumm -Z12
5. Compilerfehler

Clicker: unique_ptr als Rückgabewerte

```
unique_ptr<Z> f10(){
   unique_ptr<Z> up(new Z(4));
   cout << "f10 ";
   return up;
}
void CopyTestAuto10() {
   unique_ptr<Z> x = f10();
   cout << "Ende ";
}
```

Ausgabe?

1. +Z4 f10 Ende -Z4
2. +Z4 10 Bumm
3. +Z4 f10 +Z4 Ende -Z4 –Z4
4. +Z4 f10 Ende -Z4 –Z4

Siehe Datei WS2020_Ueb5.cpp

Clicker: unique_ptr als Rückgabewerte

```
unique_ptr<Z> f10(){
   unique_ptr<Z> up(new Z(4));
   cout << "f10 ";
   return up;
}
void WS2020_Ueb5 () {
   unique_ptr<Z> x = f10();
   cout << "Ende ";
}
```

Ausgabe?

1. +Z4 f10 Ende -Z4
2. +Z4 10 Bumm
3. +Z4 f10 +Z4 Ende -Z4 –Z4
4. +Z4 f10 Ende -Z4 –Z4

1. +Z4 f10 Ende -Z4 // Move-Semantik, da up temporäre Variable ist und ein neues Z wird sowieso nicht erzeugt
2. +Z4 10 Bumm
3. +Z4 f10 +Z4 Ende -Z4 –Z4
4. +Z4 f10 Ende -Z4 –Z4

Intelligente Zeiger als Argumente

Hieraus ergeben sich folgende Konsequenzen:
unique-Pointer müssen bei Funktionsaufrufen immer als Referenzen (oder mit move) übergeben werden (sonst Compiler-Fehler).
Ansonsten würde das Objekt in den Besitz des Funktionsarguments übergehen und würde somit beim Verlassen der Funktion zerstört.

Der Vorteil ist allerdings, dass unique-Pointer auch als Rückgabewerte (return-Werte) zurückgegeben werden können.
Die Objekte gehen am Ende der Funktion in den Besitz des Aufrufers über und werden automatisch zerstört, wenn dessen Rückgabewert zerstört wird.

Als Rückgabewerte werden unique-Pointer also durch CallByValue übergeben (keine Referenzen). Dann bleibt kein Ressourcenleck übrig.

„Eigentlich“ sollte ein Programm kein new und kein delete enthalten

```
void makeUniqueTest() {
   auto up = make_unique<int>(2011);
   // expliziter Aufruf
   // Aufruf des Kopier-Konstruktors
   auto upZ = make_unique<Z>(Z(11));
   // impiziter Aufruf des
   //Umwandlungskonstruktors
   auto upZ2 = make_unique<Z>(21);
   Z* pz = new Z(4);
}
```

```
void SimpleTest() {
   unique_ptr<Z> up(new Z(11));
   Z* pz = new Z(4);
}
```

```
Aufgabe 1
+Z11 +Z4 -Z11
```

```
Aufgabe 6
+Z11 +ZC11 -Z11 +Z21 +Z4 -Z21 -Z11
```

Intelligente Zeiger als Attribute (2)

Entsprechend ist es sinnvoll, als Elemente von Klassen nicht normale Zeiger zu verwenden, sondern Unique-Pointer. Die zugehörigen Objekte werden automatisch zerstört, ohne dass man dafür im Destruktor spezielle Vorkehrungen treffen muss

```
class X {....};

class Y
{
public:
   Y() {px=new X;}
   ~Y() {delete px;}
   Y(const Y& r) {px = new X(*r.px);}
private:
X* px;
};
```

```
#include <memory>
class X {....};      besser mit unique_ptr:

class Y
{
public:
   Y(): px(new X) {cout << "+Y";}
   ~Y() {cout << "-Y";}
    Y(const Y& ) = delete;
…
private:
unique_ptr<X> px;
};
```

Intelligente Zeiger für Arrays

Geht aber nicht, da unique_ptr nur für Speicherbereiche verwendet werden können, die mit new (nicht new[]) angefordert wurden und daher mit delete (nicht delete []) freizugeben sind.

```
class Stack
{
int * Data          ;
...};
```

besser scheint auf den ersten Blick (geht aber nicht, siehe oben):

```
class Stack
{
unique_ptr<int> Data;
...};
```

Es geht aber

```
class Stack
{
unique_ptr<int[]> Data;
...};
```

Clicker: unique_ptr und Arrays

```
class ZZ {
public:
   ZZ(int id = 9) : m_id(id) {
      cout << "+Z" << id << " ";
   }
   ~ZZ() { cout << "-Z" << m_id << " "; }
   void chVal(int j) { m_id = j; }
private:
   int m_id;
};
void WS2020_Ueb6() {
   unique_ptr<ZZ[]> up(new ZZ[3]);
   up[0].chVal(1);
   up[1].chVal(2);
   up[2].chVal(3);
   cout << "Ende ";
}
```

Ausgabe?

1. +Z9 +Z9 +Z9 Ende -Z3 -Z2 -Z1
2. +Z9 +Z9 +Z9 Ende –Z1 -Z2 –Z3
3. +Z9 +Z9 +Z9 Ende -Z3
4. +Z9 +Z9 +Z9 Ende -Z1

Siehe Datei WS2020_Ueb6.cpp

Clicker: unique_ptr und Arrays

```
class ZZ {
public:
   ZZ(int id = 9) : m_id(id) {
      cout << "+Z" << id << " ";
   }
   ~ZZ() { cout << "-Z" << m_id << " "; }
   void chVal(int j) { m_id = j; }
private:
   int m_id;
};
void WS2020_Ueb6() {
   unique_ptr<ZZ[]> up(new ZZ[3]);
   up[0].chVal(1);
   up[1].chVal(2);
   up[2].chVal(3);
   cout << "Ende ";
}
```

Ausgabe?

1. +Z9 +Z9 +Z9 Ende -Z3 -Z2 -Z1
2. +Z9 +Z9 +Z9 Ende –Z1 -Z2 –Z3
3. +Z9 +Z9 +Z9 Ende -Z3
4. +Z9 +Z9 +Z9 Ende -Z1

1. +Z9 +Z9 +Z9 Ende -Z3 -Z2 -Z1
2. +Z9 +Z9 +Z9 Ende –Z1 -Z2 –Z3
3. +Z9 +Z9 +Z9 Ende -Z3
4. +Z9 +Z9 +Z9 Ende -Z1