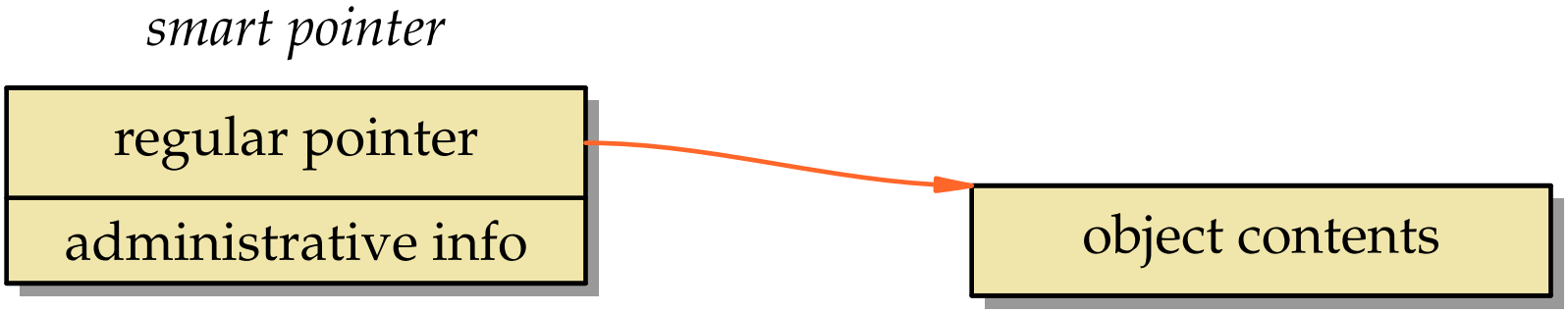


- Intelligente Zeiger (*smart pointers*) entsprechen weitgehend normalen Zeigern, haben aber Sonderfunktionalitäten aufgrund weiterer Verwaltungsinformationen.
- Sie werden insbesondere dort eingesetzt, wo die Sprache selbst keine Infrastruktur für die automatisierte Speicherfreigabe anbietet.
- Sie unterstützen das RAII-Prinzip für Zeiger.

- Seit dem C++11-Standard sind intelligente Zeiger Bestandteil der C++-Bibliothek. Zuvor gab es nur den inzwischen abgelösten *auto_ptr* und die Erweiterungen der Boost-Library, die jetzt praktisch übernommen worden sind.
- C++11 bietet folgende Varianten an:

unique_ptr	nur ein Zeiger auf ein Objekt
shared_ptr	mehrere Zeiger auf ein gemeinsames Objekt mit externem Referenzzähler
weak_ptr	nicht das Überleben sichernder „schwacher“ Zeiger auf ein Objekt mit externem Referenzzähler

- Grundsätzlich sollte ein mit **new** erzeugtes Objekt mit **delete** wieder freigegeben werden, sobald der letzte Verweis entfernt wird.
- Unterbleibt dies, haben wir ein Speicherleck.
- Wichtig ist aber auch, dass kein Objekt mehrfach freigegeben wird. Dies kann bei manueller Freigabe leicht geschehen, wenn es mehrere Zeiger auf ein Objekt gibt.
- Intelligente Zeiger können sich auch dann um eine korrekte Freigabe kümmern, wenn eine Ausnahmenbehandlung ausgelöst wird.
- Jedoch können zyklische Datenstrukturen mit der Verwendung von Referenzzählern alleine nicht korrekt aufgelöst werden. Hier sind ggf. Ansätze mit sogenannten „schwachen“ Zeigern denkbar.

- Auf ein Objekt sollten nur Zeiger eines Typs verwendet werden.
- Die einzige Ausnahme davon ist die Mischung von *shared_ptr* und *weak_ptr*.
- Im Normalfall bedeutet dies, dass die entsprechenden Klassen angepasst werden müssen, da es dann nicht mehr zulässig ist, **this** zurückzugeben.
- Üblicherweise sollte sogleich bei dem Entwurf einer Klasse geplant werden, welche Art von Zeigern zum Einsatz kommt.
- Normalerweise sollten entsprechend des RAII-Prinzips „nackte“ Zeiger außerhalb isolierter Fälle konsequent vermieden werden.

- Wenn es nur einen einzigen Zeiger auf ein Objekt geben soll, dann empfiehlt sich die Verwendung von *unique_ptr*.
- Das ist besonders geeignet für lokale Zeigervariablen oder Zeiger innerhalb einer Klasse.
- Die Freigabe erfolgt dann vollautomatisch, sobald der zugehörige Block bzw. das umgebende Objekt freigegeben werden.
- Bei einer Zuweisung wird der Besitz des Zeigers übertragen. Das funktioniert nur entsprechend mit einem sogenannten *move assignment*, d.h. der Zeigerwert wird von einem anderen *unique_ptr*-Objekt gerettet, der im nächsten Moment ohnehin dekonstruiert wird.
- Andere Zuweisungen dieser Zeiger sind nicht möglich, da dies die Restriktion des exklusiven Zugangs verletzen würde.

ptrex.cpp

```
void f(int i) {
   Object* ptr = new Object(i);
   if (i == 2) {
      throw something();
   }
   delete ptr;
}
```

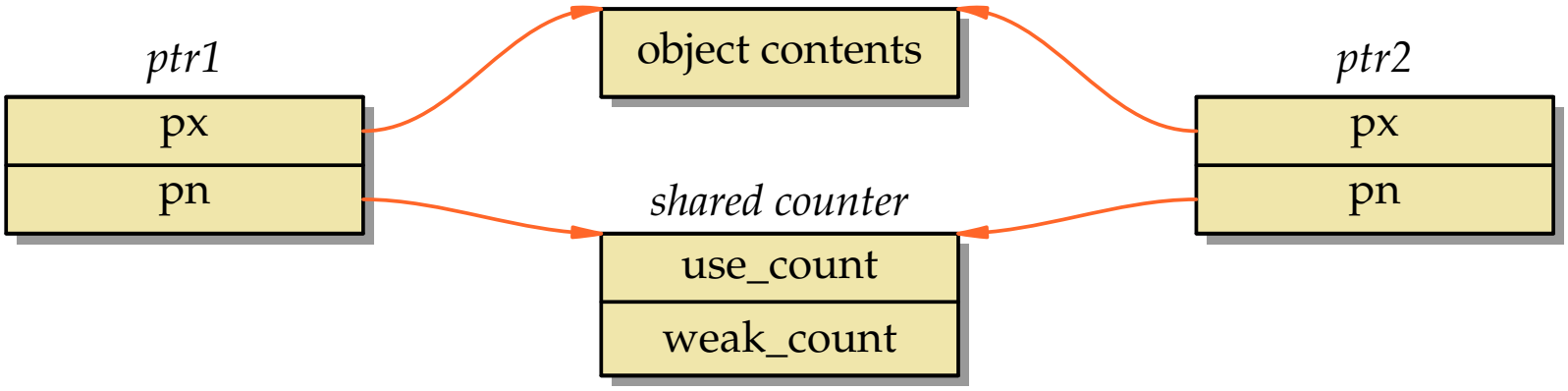
- Wenn Objekte in einer Funktion nur lokal erzeugt und verwendet werden, ist darauf zu achten, dass die Freigabe nicht vergessen wird.
- Dies passiert jedoch leicht bei Ausnahmenbehandlungen (möglicherweise durch eine aufgerufene Funktion) oder bei frühzeitigen **return**-Anweisungen.

ptrex2.cpp

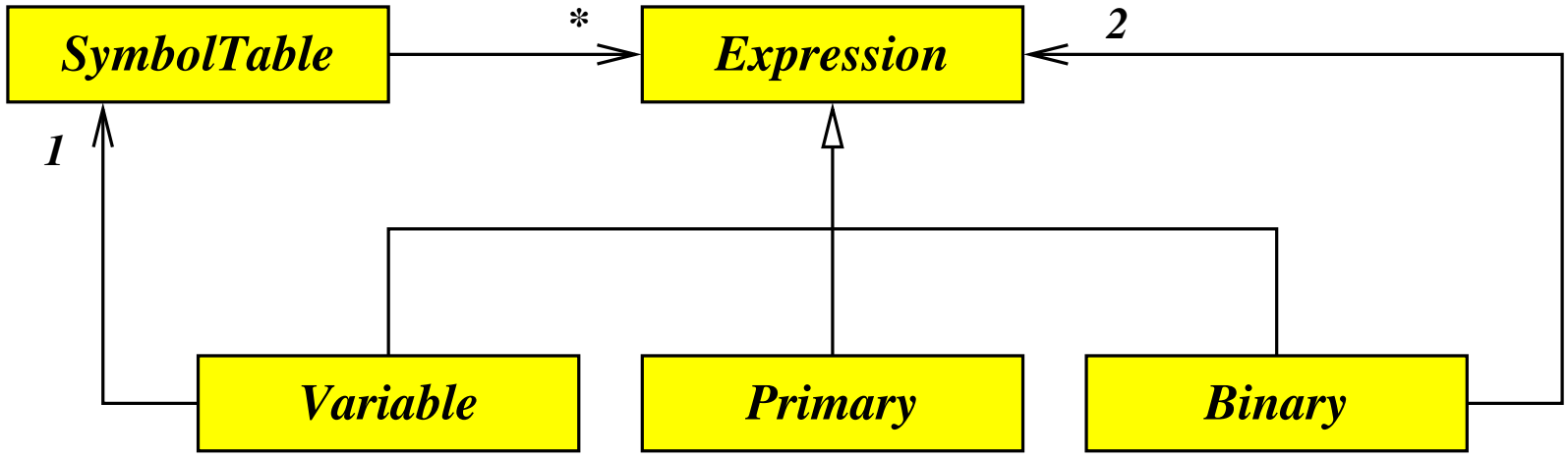
```
void f(int i) {
   unique_ptr<Object> ptr(new Object(i));
   if (i == 2) {
      throw something();
   }
}
```

- *ptr* kann hier wie ein normaler Zeiger verwendet werden, abgesehen davon, dass eine Zuweisung an einen anderen Zeiger nicht zulässig ist.
- Dann erfolgt die Freigabe des Objekts vollautomatisch über den Dekonstruktor.

Mehrere Zeiger auf ein Objekt



- Für den allgemeinen Einsatz empfiehlt sich die Verwendung von *shared_ptr*, das mit Referenzzählern arbeitet.
- Zu jedem referenzierten Objekt gehört ein intern verwaltetes Zählerobjekt, das die Zahl der Verweise zählt. Sobald *use_count* auf 0 sinkt, erfolgt die Freigabe des Objekts.
- Das Zählerobjekt wird erst freigegeben, wenn neben *use_count* auch *weak_count* auf 0 sinkt.
- Es ist darauf zu achten, dass für jedes Objekt nur ein gemeinsames Zählerobjekt existiert.



- Für einen Taschenrechner haben wir eine Datenstruktur für Syntaxbäume (*Expression*) mit den abgeleiteten Klassen *Variable*, *Primary* und *Binary*.
- Neben einem Zeiger auf den gerade eingelesenen Ausdrucksbaum kommt hier eine Symboltabelle hinzu, deren Einträge ebenfalls auf Syntaxbäume verweisen. In Abhängigkeit der Benutzereingaben können einzelne Syntaxbäume so vielfach in der Datenstruktur eingebettet sein.
- Entsprechend gibt es keinen Überblick der Verweise auf einen Syntaxbaum und somit dürfen diese erst freigegeben werden, wenn der letzte Verweis wegfällt.

expression.hpp

```
class Expression {
   public:
      virtual ~Expression() {};
      virtual Value evaluate() const = 0;
};

typedef std::shared_ptr<Expression> ExpressionPtr;
```

- Bei Klassenhierarchien, bei denen polymorphe Zeiger eingesetzt werden, ist die Deklaration eines virtuellen Dekonstruktors essentiell.
- Die Methode *evaluate* soll den durch den Baum repräsentierten Ausdruck rekursiv auswerten.
- *ExpressionPtr* wird hier als intelligenter Zeiger auf *Expression* definiert, bei dem beliebig viele Zeiger des gleichen Typs auf ein Objekt verweisen dürfen.

expression.hpp

```
class Binary: public Expression {
   public:
      typedef Value (*BinaryOp)(Value val1, Value val2);
      Binary(BinaryOp op_, ExpressionPtr expr1_,
            ExpressionPtr expr2_) :
               op{op_}, expr1{expr1_}, expr2{expr2_} {
      }
      virtual Value evaluate() const {
         return op(expr1->evaluate(), expr2->evaluate());
      }
   private:
      BinaryOp op;
      ExpressionPtr expr1;
      ExpressionPtr expr2;
};
```

- *Binary* repräsentiert einen Knoten des Syntaxbaums mit einem binären Operator und zwei Operanden.
- Statt *Expression*∗ wird dann konsequent *ExpressionPtr* verwendet.

Hierarchie der intelligenten Zeiger

expression.hpp

```
class Variable: public Expression {
   public:
      Variable(SymbolTable& symtab_, const std::string& varname_) :
         symtab{symtab_}, varname{varname_} {
      }
      virtual Value evaluate() const {
         ExpressionPtr expr = symtab.get(varname);
         return expr? expr->evaluate(): Value();
      }
      void set(ExpressionPtr expr) {
         symtab.define(varname, expr);
      }
   private: SymbolTable& symtab; std::string varname;
};
typedef std::shared_ptr<Variable> VariablePtr;
```

- Die Kompatibilität innerhalb der *Expression*-Hierarchie überträgt sich auch auf die zugehörigen intelligenten Zeiger. Zwar bilden die intelligenten Zeigertypen keine formale Hierarchie, aber sie bieten Zuweisungs-Operatoren auch für fremde Datentypen an, die nur dann funktionieren, wenn die Kompatibilität für die entsprechenden einfachen Zeigertypen existiert.

parser.cpp

```
ExpressionPtr Parser::parseSimpleExpression() throw(Exception) {
   ExpressionPtr expr = parseTerm();
   while (getToken().symbol == Token::PLUS ||
          getToken().symbol == Token::MINUS) {
      Binary::BinaryOp op;
      switch (getToken().symbol) {
         case Token::PLUS: op = addop; break;
         case Token::MINUS: op = subop; break;
         default: /* does not happen */ break;
      }
      nextToken();
      ExpressionPtr expr2 = parseTerm();
      expr = std::make_shared<Binary>(op, expr, expr2);
   }
   return expr;
}
```

- *make_shared* erzeugt ein Objekt des angegebenen Typs mit **new** und liefert den passenden intelligenten Zeigertyp zurück.
- Das ist in diesem Beispiel *shared_ptr<Binary>*, das entsprechend der Klassenhierarchie an den allgemeinen Zeigertyp *ExpressionPtr* zugewiesen werden kann.

Typentests und Konvertierungen bei intelligenten Zeigern

parser.cpp

```
ExpressionPtr Parser::parseAssignment() throw(Exception) {
   ExpressionPtr expr = parseSimpleExpression();
   if (getToken().symbol == Token::BECOMES) {
      VariablePtr var = std::dynamic_pointer_cast<Variable>(expr);
      if (!var) {
         throw Exception(getToken(), "variable expected");
      }
      nextToken();
      ExpressionPtr expr2 = parseSimpleExpression();
      var->set(expr2);
      return expr2;
   }
   return expr;
}
```

- Statt **dynamic_cast** ist bei intelligenten Zeigern *dynamic_pointer_cast* zu verwenden, um eine ungewollte Neu-Erzeugung eines Zählerobjekts zu vermeiden.
- Genauso wie bei **dynamic_cast** wird ein Nullzeiger geliefert, falls der angegebene Zeiger nicht den passenden Typ hat.

- Bei Referenzzyklen bleiben die Referenzzähler positiv, selbst wenn der Zyklus insgesamt nicht mehr von außen erreichbar ist.
- Eine automatisierte Speicherfreigabe (*garbage collection*) würde den Zyklus freigeben, aber mit Zeigern auf Basis von *shared_ptr* gelingt dies nicht.
- Eine Lösung für dieses Problem sind sogenannte schwache Zeiger (*weak pointers*), die bei der Referenzzählung nicht berücksichtigt werden.

Beispiel: Doppelt verkettete Liste

list.hpp

```
template <typename T>
class List {
   private:
      struct Element;
      typedef std::shared_ptr<Element> Link;
      typedef std::weak_ptr<Element> WeakLink;
      struct Element {
         Element(const T& elem_);
         T elem;
         Link next;
         WeakLink prev;
      };
      Link head;
      Link tail;
   public:
      class Iterator {
         // ...
      };
      Iterator begin();
      Iterator end();
      void push_back(const T& object);
};
```

Beispiel: Doppelt verkettete Liste

list.hpp

```
typedef std::shared_ptr<Element> Link;
typedef std::weak_ptr<Element> WeakLink;
struct Element {
   Element(const T& elem_);
   T elem;
   Link next;
   WeakLink prev;
};
```

- Die einzelnen Glieder einer doppelt verketteten Liste verweisen jeweils auf den Nachfolger und den Vorgänger.
- Wenn mindestens zwei Glieder in einer Liste enthalten ist, ergibt dies eine zyklische Datenstruktur.
- Das kann dadurch gelöst werden, dass für die Rückverweise schwache Zeiger verwendet werden.

Beispiel: Doppelt verkettete Liste

list.tpp

```
template<typename T>
void List<T>::push_back(const T& object) {
   Link ptr = std::make_shared<Element>(object);
   ptr->prev = tail;
   if (head) {
      tail->next = ptr;
   } else {
      head = ptr;
   }
   tail = ptr;
}
```

- Eine Zuweisung von *shared_ptr* an den korrespondierenden *weak_ptr* ist problemlos möglich wie hier bei: $ptr->prev = tail$

Iterator-Klasse mit schwachem Zeiger

list.hpp

```
class Iterator {
   public:
      class Exception: public std::exception {
         public:
            Exception(const std::string& msg_);
            virtual ~Exception() noexcept;
            virtual const char* what() const noexcept;
         private:
            std::string msg;
      };
      bool valid();
      T& operator*();
      Iterator& operator++(); // prefix increment
      Iterator operator++(int); // postfix increment
      Iterator& operator--(); // prefix decrement
      Iterator operator--(int); // postfix decrement
      bool operator==(const Iterator& other);
      bool operator!=(const Iterator& other);
   private:
      friend class List;
      Iterator();
      Iterator(WeakLink ptr_);
      WeakLink ptr;
};
```

Dereferenzierung eines schwachen Zeigers

list.tpp

```
template<typename T>
T& List<T>::Iterator::operator*() {
   Link p = ptr.lock();
   if (p) {
      return p->elem;
   } else {
      throw Exception("iterator is expired");
   }
}
```

- Ein schwacher Zeiger kann mit Hilfe der *lock*-Methode in einen regulären Zeiger verwandelt werden.
- Wenn das referenzierte Objekt mittlerweile freigegeben wurde, ist der Zeiger 0.

list.tpp

```
template<typename T>
bool List<T>::Iterator::operator==(const Iterator& other) {
   Link p1 = ptr.lock();
   Link p2 = other.ptr.lock();
   return p1 == p2;
}

template<typename T>
bool List<T>::Iterator::operator!=(const Iterator& other) {
   return !(*this == other);
}
```

- Schwache Zeiger können erst dann miteinander verglichen werden, wenn sie zuvor in reguläre Zeiger konvertiert werden.
- Nullzeiger werden hier als äquivalent angesehen.

```
#include <memory>

class Object;
typedef std::shared_ptr<Object> ObjectPtr;
class Object: public std::enable_shared_from_this<Object> {
   public:
      ObjectPtr me() {
         return shared_from_this();
      }
};
```

- Die Grundregel, dass auf ein Objekt nur Zeiger eines Typs verwendet werden sollten, stößt auf ein Problem, wenn statt **this** ein passender intelligenter Zeiger zurückzugeben ist.
- Eine Lösung besteht darin, die Klasse von *std*::*enable_shared_from_this* abzuleiten. Dann steht die Methode *shared_from_this* zur Verfügung. Dies wird implementiert, indem im Objekt zusätzlich ein schwacher Zeiger auf das eigene Objekt verwaltet wird.

```
#include <memory>

class Object;
typedef std::shared_ptr<Object> ObjectPtr;
class Object {
   public:
      class Key {
         friend class Object;
         Key() {}
      };
      static ObjectPtr create() {
         return std::make_shared<Object>(Key());
      }
      Object(Key&& key) {}
};
```

- Um die Grundregel durchzusetzen, erscheint es gelegentlich sinnvoll, die regulären Konstruktoren zu verbergen.
- **private** dürfen Sie jedoch nicht sein, da *std*::*make_shared* einen passenden öffentlichen Konstruktor benötigt.
- Eine Lösung bietet der *pass key*-Ansatz. Der Konstruktor ist zwar öffentlich, aber ohne privaten Schlüssel nicht benutzbar.