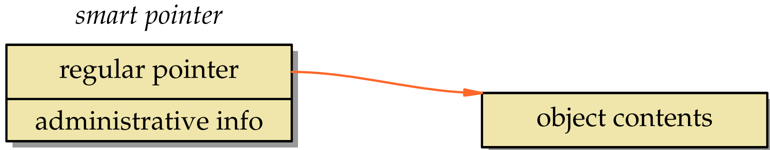


- Intelligente Zeiger (*smart pointers*) entsprechend weitgehend normalen Zeigern, haben aber Sonderfunktionalitäten aufgrund weiterer Verwaltungsinformationen.
- Sie werden insbesondere dort eingesetzt, wo die Sprache selbst keine Infrastruktur für die automatisierte Speicherfreigabe anbietet.

- Der bisherige C++-Standard bietet bislang kaum Unterstützung (mit der Ausnahme von *auto_ptr*, das wahrscheinlich im nächsten Standard abgelöst werden wird).
- Federführend bei der Entwicklung von intelligenten Zeigern in C++ ist die Boost-Library. Teile davon fanden bereits Eingang in TR1 (ISO/IEC TR 19768), das der Vorbereitung des nächsten Standards für C++ dient.
- Die Boost-Library bietet folgende Varianten an:

scoped_ptr	nur ein Zeiger auf ein Objekt
shared_ptr	mehrere Zeiger auf ein Objekt mit externem Referenzzähler
weak_ptr	nicht das Überleben sichernder „schwacher“ Zeiger auf ein Objekt mit externem Referenzzähler
intrusive_ptr	mehrere Zeiger auf ein Objekt mit integriertem Referenzzähler

Ferner gibt es noch *scoped_array* und *shared_array* für dynamische Arrays.

- 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 Ausnahmebehandlungen 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 entsprechend 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.

- Wenn es nur einen einzigen Zeiger auf ein Objekt geben soll, dann empfiehlt sich die Verwendung von *scoped_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.
- Eine Zuweisung dieser Zeiger ist nicht möglich, da dies die Restriktion des exklusiven Zugangs verletzen würde.
- *scoped_ptr* kann nicht als Element-Typ eines Containers verwendet werden, da Zuweisungen hier zwingend notwendig sind.

ptrex.cpp

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

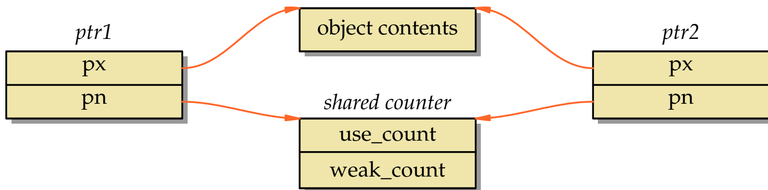
- Wenn Objekte in einer Funktion nur lokal erzeugt und verwendet werden, ist darauf zu achten, dass die Freigabe nicht vergessen wird.
- Dies passiert jedoch leicht bei Ausnahmenbehandlungen (möglicherweise durch eine aufgerufene Funktion) oder bei frühzeitigen **return**-Anweisungen.

Lokale Zeiger: Einsatz von *scoped_ptr*

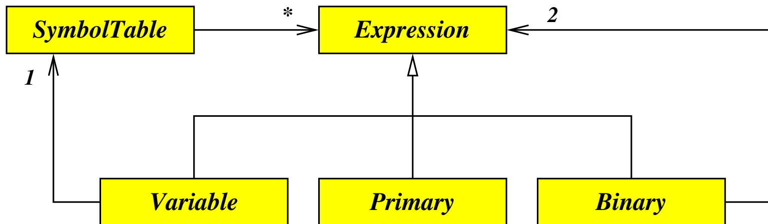
ptrex2.cpp

```
void f(int i) {
   scoped_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.



- 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.



- Für einen Taschenrechner haben wir eine Datenstruktur für Syntaxbäume (*Expression*) mit den abgeleiteten Klassen *Variable*, *Primary* und *Binary*.
- Einträge in der Symboltabelle können auch auf Syntaxbäume verweisen.

expression.hpp

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

typedef boost::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);
      virtual Value evaluate() const;
   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.

expression.hpp

```
class Variable: public Expression {
   public:
      Variable(SymbolTable& _symtab, const std::string& _varname);
      virtual Value evaluate() const;
      void set(ExpressionPtr expr);
   private:
      SymbolTable& symtab;
      std::string varname;
};
typedef boost::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.

Intelligente Zeiger auf neuerzeugte Objekte

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 = add; break;
         case Token::MINUS: op = sub; break;
      }
      nextToken();
      ExpressionPtr expr2 = parseTerm();
      expr = make_shared<Binary>(op, expr, expr2);
   }
   return expr;
}
```

- *make_shared*, das ebenfalls zur Boost-Library gehört, 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 = 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 sicherzustellen dass es bei einem Zählerobjekt bleibt.
- 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.

list.hpp

```
template <typename T>
class List {
   private:
      struct Element;
      typedef boost::shared_ptr<Element> Link;
      typedef boost::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);
};
```

list.hpp

```
typedef boost::shared_ptr<Element> Link;
typedef boost::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.

list.cpp

```
template<typename T>
void List<T>::push_back(const T& object) {
   Link ptr = boost::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: $ptr{-}{>}prev = tail$

list.hpp

```
class Iterator {
   public:
      class Exception: public std::exception {
         public:
            Exception(const std::string& _msg);
            virtual ~Exception() throw();
            virtual const char* what() const throw();
         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;
};
```

list.cpp

```
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.

Vergleich von schwachen Zeigern

list.cpp

```
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.