Beispiele

- Objekte speichern: `Storable` und `Storage`
- Ein Binärbaum

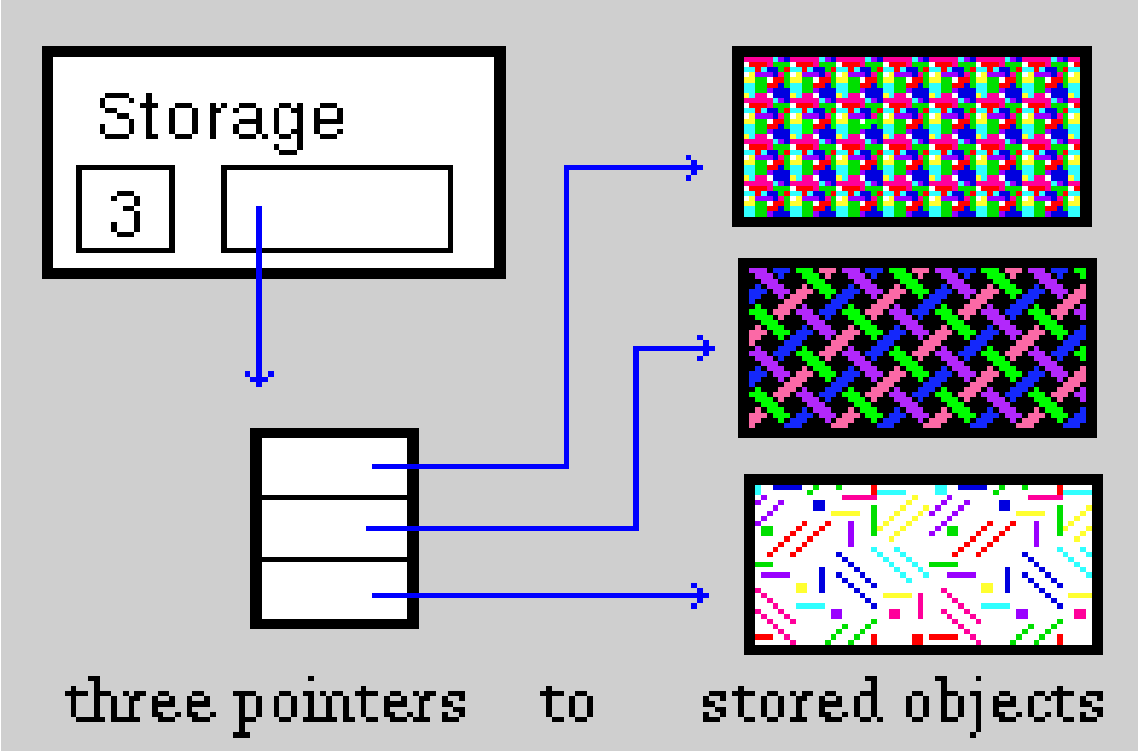
Beispiele

Objekte speichern: `Storable` und `Storage`

`Storable` dient als Prototyp für Objekte, die gespeichert werden können

`Storage` dient zum Speichern von Objekten

`Storage` verwaltet ein dynamisch allokiertes Feld von Zeigern auf die gespeicherten Objekte



Objekte speichern: `Storable` und `Storage`

Schnittstelle der Klasse `Storage`

`void add(Storable const *newobj)`

speichert ein Objekt.

`Storable *get(int index)`

liefert einen Zeiger auf dasjenige Objekt, das an der `index`-ten Stelle im Speicher abgelegt ist.

`int nstored() const`

liefert die Anzahl der gespeicherten Objekte.

Objekte speichern: `Storable` und `Storage`

Kopieren oder nicht?

Sollten Kopien der Originalobjekte gespeichert werden oder die Objekte selbst?

```
Storage store;
Storable something;

store.add(something);      // add to storage

// assume that Storable::modify() is defined
something.modify();
// modify original object

Storable *retrieved = store.get(0);
// retrieve from storage

// NOW: is "*retrieved" equal to "something" ?!
```

Wir speichern Kopien der Originalobjekte.

Objekte speichern: `Storable` und `Storage`

Wer fertigt die Kopie an?

Die Klasse `Storage` soll möglichst universell einsetzbar sein. Daher kann die Kopie nicht in der Klasse `Storage` angefertigt werden. Der tatsächliche Typ der zu speichernden Objekte ist nicht im Voraus bekannt.

```
void Storage::add(Storable const *obj)
{
  Storable *to_store = new Storable(*obj);
  // now add to_store instead of obj
  // ...
}
```

`Storable` ist nur eine Basisklasse und kann daher keine Objekte einer abgeleiteten Klasse anlegen.

Die Kopie muss also in der abgeleiteten Klasse angefertigt werden. Eine solche Klasse muss eine Kopie eines Objekts anlegen und einen Zeiger auf das Duplikat zurückgeben können.

```
void Storage::add(Storable const *obj)
{
  Storable *to_store = obj->duplicate();
  // now add to_store instead of obj
  // ...
}
```

Objekte speichern: `Storable` und `Storage`

Die Klasse `Storable`

- Braucht `Storable` Konstruktoren?

- Braucht `Storable` einen Destruktor?
  Sollte dieser virtuell oder gar rein virtuell sein?

```
class Storable
{
  public:
    virtual ~Storable();
    virtual Storable *duplicate() const = 0;
};

Storable::~Storable()
{}
```

Objekte speichern: `Storable` und `Storage`

Konvertieren eines Objekts einer existierenden Klasse in ein `Storable`

```
class Person: public Storable
{
  public:
    // default constructor
    Person(void);

    // copy constructor
    Person(Person const &other);

    // assignment
    Person const &operator=
      (Person const &other);

    // duplicator function
    Person *duplicate() const;

    // ...
};
```

Objekte speichern: `Storable` und `Storage`

Konvertieren eines Objekts einer existierenden Klasse in ein `Storable`

```
// first version:
Person *Person::duplicate1() const
{
  // uses default constructor in new Person
  Person *dup = new Person;

  // uses overloaded assignment in *dup = *this
  *dup = *this;

  return dup;
}
```

```
// second version:
Person *Person::duplicate() const
{
  // uses copy constructor in new Person(*this)
  return new Person(*this);
}
```

Objekte speichern: `Storable` und `Storage`

Konvertieren eines Objekts einer existierenden Klasse in ein `Storable`

```
class StorablePerson :
  public Person, public Storable
{
  public:
    // copy constructor
    StorablePerson(StorablePerson const &other);

    // duplicator function
    StorablePerson *duplicate() const;

    // ...
};

StorablePerson *StorablePerson::duplicate
  () const
{
  return new StorablePerson(*this);
}
```

Objekte speichern: `Storable` und `Storage`

Die Klasse `Storage`

```
class Storage: public Storable
{
  public:
    // constructors, destructor
    Storage();
    Storage(Storage const &other);
    ~Storage();

    // overloaded assignment
    Storage const &operator=
      (Storage const &other);

    // functionality to duplicate storages
    Storage *duplicate() const;

    // interface
    void add(Storable const *newobj);
    Storable *get(int index);
    int nstored() const;

  private:
    // copy/destroy primitives
    void destroy();
    void copy(Storage const &other);

    // private data
    int n;
    Storable **storage;
};
```

Objekte speichern: `Storable` und `Storage`

Die Klasse `Storage`

```
// default constructor
Storage::Storage()
{
  n = 0;
  storage = 0;
}

// copy constructor
Storage::Storage(Storage const &other)
{
  copy(other);
}

// destructor
Storage::~Storage()
{
  destroy();
}

// use copy constructor to duplicate
// the current object
Storage *Storage::duplicate() const
{
  return new Storage(*this);
}
```

Objekte speichern: `Storable` und `Storage`

Die Klasse `Storage`

```
// overloaded assignment
Storage const &Storage::operator=
  (Storage const &other)
{
  if (this != &other)
  {
    destroy();
    copy(other);
  }
  return *this;
}

void Storage::copy(Storage const &other)
{
  n = other.n;
  storage = new Storable* [n];
  for (int i = 0; i < n; i++)
    storage [i] =
      other.storage [i]->duplicate();
}

void Storage::destroy()
{
  for (register int i = 0; i < n; i++)
    delete storage [i];
  delete storage;
}
```

Objekte speichern: `Storable` und `Storage`

Die Klasse `Storage`

```
void Storage::add(Storable const *newobj)
{
  // reallocate storage array
  storage = (Storable **)
    realloc(storage,
            (n + 1) * sizeof(Storable *));
  // put duplicate of newobj in storage
  storage [n] = newobj->duplicate();
  // increase number of obj in storage
  n++;
}

Storable *Storage::get(int index)
{
  // check if index within range
  if (index < 0 || index >= n)
    return 0;
  // return address of stored object
  return storage [index];
}

int Storage::nstored() const
{
  return n;
}
```

Beispiele

Ein Binärbaum

Die Klasse `Node`

`Node` ist eine abstrakte (rein virtuelle) Klasse, die das Protokoll für die Verwendung abgeleiteter Klassen in einem `Tree` definiert.

Schnittstelle der Klasse `Node`

```
virtual int compare
  (Node const *other) const = 0;
```

dient zum Vergleich zweier Knoten im Baum.

```
virtual Node* duplicate() const = 0;
```

wird benötigt, weil der Binärbaum *Kopien* von Objekten enthält.

```
virtual void process() = 0;
```

Beim Baumdurchlauf wird auf jedem Knoten eine vom Typ des Knotens abhängige Operation ausgeführt.

```
virtual void already_stored();
```

Ein Objekt wird nur *einmal* im Baum gespeichert.

Ein Binärbaum

Die Klasse Node

```
class Node
{
  public:
    // destructor
    virtual ~Node();

    // duplicator
    virtual Node* duplicate() const = 0;

    // comparison of 2 objects
    virtual int compare
      (Node const *other) const = 0;

    // function to do whatever is needed
    // to the node
    virtual void process() = 0;

    // called when object to add was already
    // in the tree
    virtual void already_stored();
};

Node::~Node()
{}

void Node::already_stored()
{}
```

Ein Binärbaum

Die Klasse Tree

```
class Tree
{
  public:
    // constructors, destructor
    Tree();
    Tree(Tree const &other);
    ~Tree();

    // assignment
    Tree const &operator=(Tree const &other);

    // addition of a Node
    void add(Node *what);

    // processing order in the tree
    void preorder_walk();
    void inorder_walk();
    void postorder_walk();

  private:
    // primitives
    void copy(Tree const &other);
    void destroy();

    // data
    Tree *left, *right;
    Node *info;
};
```

Ein Binärbaum

Die "Standard"-Funktionen

```
// default constructor: initializes to 0
Tree::Tree()
{
  left = right = 0;
  info = 0;
}

// copy constructor
Tree::Tree(Tree const &other)
{
  copy(other);
}

// destructor: destroys the tree
Tree::~Tree()
{
  destroy();
}

// overloaded assignment
Tree const &Tree::operator=(Tree const &other)
{
  if (this != &other)
  {
    destroy();
    copy(other);
  }
  return *this;
}
```

Ein Binärbaum

Einfügen eines Objekts in den Binärbaum

```
void Tree::add(Node *what)
{
  if (! info)
    info = what->duplicate();
  else
  {
    register int cmp = info->compare(what);

    if (cmp < 0)
    {
      if (! left)
      {
        left = new Tree;
        left->info = what->duplicate();
      }
      else
        left->add(what);
    }
    else if (cmp > 0)
    {
      if (! right)
      {
        right = new Tree;
        right->info = what->duplicate();
      }
      else
        right->add(what);
    }
    else
      info->already_stored();
  }
}
```

Ein Binärbaum

Durchsuchen des Binärbaums

```
void Tree::preorder_walk()
{
  if (info)
    info->process();
  if (left)
    left->preorder_walk();
  if (right)
    right->preorder_walk();
}

void Tree::inorder_walk()
{
  if (left)
    left->inorder_walk();
  if (info)
    info->process();
  if (right)
    right->inorder_walk();
}

void Tree::postorder_walk()
{
  if (left)
    left->postorder_walk();
  if (right)
    right->postorder_walk();
  if (info)
    info->process();
}
```

Ein Binärbaum

Die Operationen copy() und destroy()

```
void Tree::destroy()
{
  delete info;
  if (left)
    delete left;
  if (right)
    delete right;
}

void Tree::copy(Tree const &other)
{
  info = other.info ?
    other.info->duplicate() : 0;
  left = other.left ?
    new Tree(*other.left) : 0;
  right = other.right ?
    new Tree(*other.right) : 0;
}
```

Ein Binärbaum

Verwendung von `Tree` und `Node`:

in Textdateien enthaltene Worte zählen

```
class StrNode: public Node
{
  public:
    // constructors, destructor
    StrNode();
    StrNode(StrNode const &other);
    StrNode(char const *s);
    ~StrNode();

    // assignment
    StrNode const &operator=
      (StrNode const &other);

    // functions required by Node protocol
    StrNode* duplicate() const;
    int compare(Node const *other) const;
    void process();
    void already_stored();

  private:
    // data
    char *str;
    int times;
};
```

Ein Binärbaum

Die Klasse `StrNode`

```
StrNode::StrNode()
{
  str = 0;
  times = 0;
}

StrNode::StrNode(StrNode const &other)
{
  str = strdupnew(other.str);
  times = other.times;
}

StrNode::StrNode(char const *s)
{
  str = strdupnew(s);
  times = 1;
}

StrNode::~StrNode()
{
  delete [] str;
}
```

Ein Binärbaum

Die Klasse `StrNode`

```
StrNode const &StrNode::operator=
  (StrNode const &other)
{
  if (this != &other)
  {
    delete [] str;
    str = strdupnew(other.str);
    times = other.times;
  }
  return *this;
}

int StrNode::compare(Node const *other) const
{
  StrNode *otherp = (StrNode *) other;

  if (str && otherp->str)
    return strcmp(str, otherp->str);

  if (! str && ! otherp->str)
    return 0;

  return (int) otherp->str - (int) str ;
}
```

Ein Binärbaum

Die Klasse `StrNode`

```
StrNode *StrNode::duplicate() const
{
  return new StrNode(*this);
}

void StrNode::process()
{
  if (str)
    printf("%4d\t%s\n", times, str);
}

void StrNode::already_stored()
{
  times++;
}
```

Ein Binärbaum

Verwendung von `Tree` und `Node`:

in Textdateien enthaltene Worte zählen

```
void countfile(FILE *inf, char const *name)
{
   Tree tree;
   char buf [255];

   while (true)
   {
      fscanf(inf, " %s", buf);
      if (feof(inf))
         break;

      StrNode *word = new StrNode(buf);

      tree.add(word);
      delete word;
   }
   tree.inorder_walk();
}
```

Ein Binärbaum

Verwendung von `Tree` und `Node`:

in Textdateien enthaltene Worte zählen

```
int main(int argc, char **argv)
{
  register int exitstatus = 0;

  if (argc > 1)
    for (register int i = 1; i < argc; i++)
    {
      FILE *inf = fopen(argv [i], "r");

      if (! inf)
      {
        fprintf(stderr,
                "wordc: can't open \"%s\"\n",
                argv [i]);
        exitstatus++;
      }
      else
      {
        countfile(inf, argv [i]);
        fclose(inf);
      }
    }
  else
    countfile(stdin, "--stdin--");
  return exitstatus;
}
```