Namensräume

Inhalt

1. Namenskonventionen und Dokumentationsrichtlinien
2. Eigenschaften von Namensräumen
3. Objekträume und Speichermodelle
4. Allgemeine Beschreibungssprache für Softwarebauelemente
5. Verarbeitungsmodelle
6. Beispiele für Modultypen in C/C++

Namensräume

1. Namenskonventionen und Dokumentationsrichtlinien

- Namen von Konstanten werden durchgängig groß geschrieben, z.B.:

```
const int NUMBER = 100;
```

- Namen von Nichtstandardtypen erhalten das Präfixsymbol T, z.B.:

```
typedef struct {
  ...
} TPerson;
```

- Modulnamen erhalten das Präfixsymbol `M`, z.B.: `MPerson` für das Modul Person

- Namen von Klassen erhalten das Präfixsymbol `C`, z.B.:

```
class CPerson {
  ...
};
```

- Namen von Interfaces erhalten das Präfixsymbol `I`.

Namensräume

1. Namenskonventionen und Dokumentationsrichtlinien

- Namen von Pointer-Typen erhalten das Präfixsymbol `P`, z.B.:

```
typedef TPerson* PPerson;
```

  Ein Sonderfall ist der `void` Pointer; er erhält das Präfixsymbol `U` (untypisiert), z.B.:

```
typedef void* UPerson;
```

- Eine Übersetzungseinheit kann durch einen Kopfkommentar kommentiert werden, der
  - die Funktion des Bausteins,
  - den Namen der Datei,
  - einen Autorenhinweis,

  enthalten soll, z.B.:

Namensräume

1. Namenskonventionen und Dokumentationsrichtlinien

Template für den Dokumentationskopf einer Übersetzungseinheit

```
/* <Task of the computation unit>                  */
/* File name: <name of the file>                   */
/* Author: <name>   ;  Date: <date of creation>    */
/* Version: <number of version> ; Update: <date>   */
```

Beispiel für den Dokumentationskopf einer Übersetzungseinheit in C++

```
/* Module: Person                                  */
/* Name of file: PERSON.h                          */
/* Author: XXX              Date: 1.10.2010        */
/* Version: 1.0.1           Update: 12.10.2010     */
```

Namensräume

1. Namenskonventionen und Dokumentationsrichtlinien

Jede Funktion kann durch einen Kopfkommentar kommentiert werden, z.B.:

```
bool InitDate(TDate* dat, unsigned int da,
              unsigned int mo, unsigned int ye);
/* Initializes the attributes of a date         */
```

oder erweitert

```
bool InitDate(TDate* dat, unsigned int da,
              unsigned int mo, unsigned int ye);
/* Initializes the attributes of a date dat
/* with day(da), month(mo), year(ye)           */
/* pre:  dat is created                        */
/* post: dat is initialized, returns true or   */
/*       dat is not initialized, returns false */
```

Es ist auch die ungarische Notation von Funktionsnamen möglich (Java).

```
bool initDate(TDate* dat, unsigned int da,
              unsigned int mo, unsigned int ye);
```

Namensräume

2. Eigenschaften von Namensräumen

Glossar

Namensraum (namespace)

Ein Namensraum ist eine Menge von Namen, die miteinander in Beziehung stehen können. Ein Namensraum hat selbst einen Namen. Ein Namensraum kann Bestandteil eines Namensraumes sein (Einbettung).

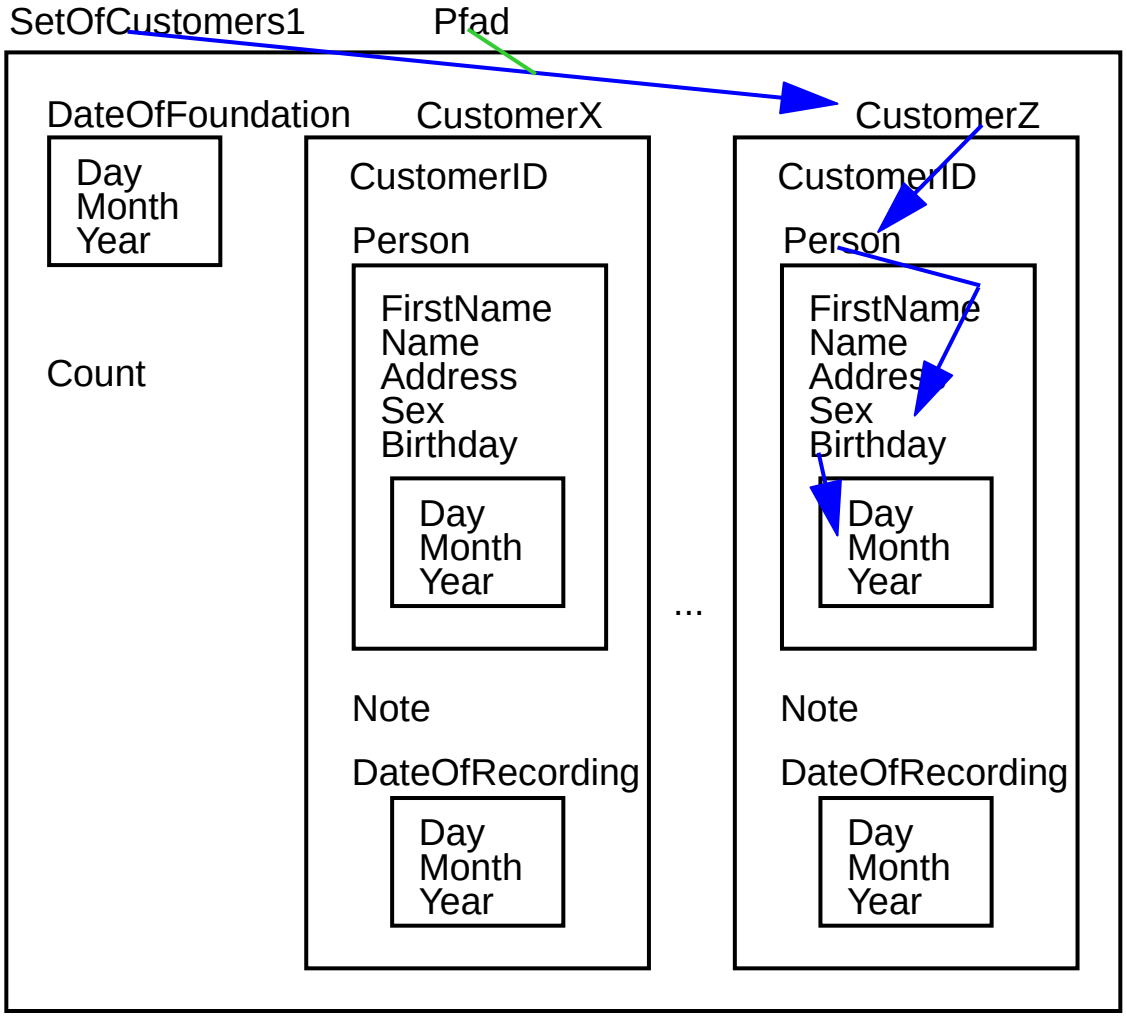
Elementare und komposite Namen

- Elementare Namen bezeichnen programmiersprachliche Objekte oder Namensräume.
- Komposite Namen sind zusammengesetzte Namen, die aus Sequenzen elementarer Namen und deren Verknüpfung durch einen Verbindungsoperator, z.B. “.“, bestehen.
- Um eine Eindeutigkeit in der Namensraumstruktur zu erreichen, werden Pfade eingeführt. Ein Pfad ist eine Sequenz von elementaren Namen.
- Komposite Namen lassen sich für das Beispiel als Pfade durch Namensräume interpretieren.

Namensräume

2. Eigenschaften von Namensräumen

Beispiel für Namensräume und Pfade



Pfad: SetOfCustomers1.CustomerZ.Person.Birthday.Month

Namensräume

2. Eigenschaften von Namensräumen

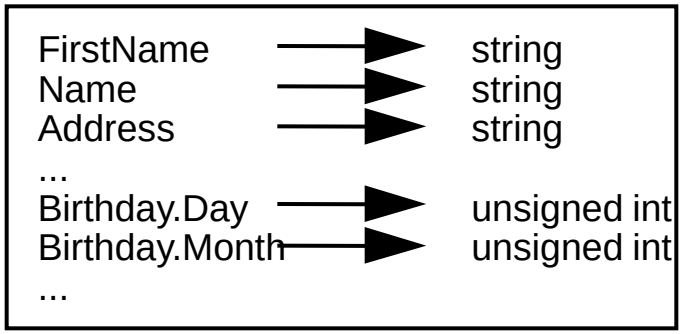
Glossar

Attributierungsbeziehung

- Eine Attributierungsrelation setzt elementare oder komposite Namen und Attribute in Beziehung.
- Ein Attribut ist wieder ein elementarer oder kompositer Name.

Attributierung von Namen eines Namensraumes

Person



Namensräume

2. Eigenschaften von Namensräumen

Beispiele für Namensräume

- Alle Namen, die in einem Deklarationsniveau deklariert sind (z.B. einer Prozedur, einem Modul usw.).

- Alle Namen, die in einem Deklarationsniveau verwendet werden können (in einer Prozedur, in einem Modul, in einer Klasse). Diese Namensräume bestehen aus den Namen des entsprechenden Deklarationsniveaus und Namen aus der Umgebung (z.B. nicht verdeckte globale Namen bei Prozeduren oder importierte Namensräume bei Moduln).

- Nutzerdefinierte logische Gruppierungen von Namen, wie z.B. durch namespace-Deklarationen in C++, z.B. Namensraum KeyAttr der Schlüsselattribute von Person.

```
namespace KeyAttr {
  string FirstName;
  string Name;
  string Address
}
```

Namensräume

2. Eigenschaften von Namensräumen

Explizite und implizite Namensräume von Softwarebauelementen.

- Implizite Namensräume werden durch das Blockkonzept realisiert.
- Explizite Namensräume basieren auf dem „namespaceKonzept“.

Namensräume

2. Eigenschaften von Namensräumen

Namensräume bei der Programmierung im Kleinen

Glossar

Implizite Namensräume

- Ein Block ist eine Programmeinheit, die aus einer Folge von Deklarationen und Anweisungen besteht. Blöcke können mit **`begin`** und **`end`** geklammert sein. Das Environment eines Blocks wird durch andere Blöcke gebildet. Blöcke können geschachtelt werden.

- Es gelten folgende Regeln für die Gültigkeit und Sichtbarkeit von Namen:
  - Alle Namen, die in einem Block deklariert werden, sind nur in diesem Block sichtbar.
  - Namen, die außerhalb eines Blocks deklariert sind, erhalten eine neue Bedeutung, wenn sie innerhalb dieses Blocks neu deklariert werden.
  - Namen, die in übergeordneten Blöcken deklariert wurden, sind, falls sie nicht innerhalb eines Blocks umdeklariert wurden, in diesem Block sichtbar.

Namensräume

2. Eigenschaften von Namensräumen

Namensräume bei der Programmierung im Kleinen

Ein Beispiel für eine Blockstruktur

```
…
int x,y;              // global variables
const int A = 20;     // global constant
int Func1(int x, int z)
{
    return x + z + A;
}
int Func2(int y, int z)
{
    return y - z - x;
}
void Show(const char* s, int z)
{
    cout << s << z << endl;
}
```

```
int main(void)
{
    // local namespace hides global namespace
    const int B = 10;     // local constante
    int x;                // local variable
    x = Func1(A, B);
    y = Func2(B, x);
    Show("x = ", x);      // shows local x
    Show("y = ", y);      // shows global y
    return 0;
}
```

Namensräume

2. Eigenschaften von Namensräumen

Namensräume bei der Programmierung im Kleinen

Ein Beispiel für eine Blockstruktur

B0

```
...
int x,y;                  // global variables
const int A = 20;         // global constant
int Func1(int x, int z)
   B1 {
      return x + z + A;
      }
int Func2 (int y, int z)
   B2 {
      return y - z - x;
      }
void Show(const char* s,int z)
   B3 {
      cout << s << z << endl;
      }
int main(void)
   B4 {
      // local namespace hides global namespace
      const int B = 10;  // local constant
      int x;             // local variable, hides global x
      x = Func1(A, B);
      y = Func2(B, x);
      Show("x = ", x);  // shows local x
      Show("y = ", y);  // shows global y
      return 0;
      }
```

Namensräume

2. Eigenschaften von Namensräumen

Beispiele für explizite Namensräume

```
namespace data {
  int x;
  int y;
}
```

Der Zugriff auf Namen aus diesem Namensraum kann auf verschiedene Art erfolgen:

- qualifiziert:

```
data::x = 33
```

Dieser Zugriff ist für alle Blöcke möglich.

- **using**-Deklaration

```
...
using data::x;     // import x from namespace data
x = 33;            // use imported name x
data::y = 22;      // use non-imported name y
...
```

Namensräume

2. Eigenschaften von Namensräumen

Beispiele für explizite Namensräume

- **`using`-**Directive

```
...
using namespace data;
x = 33;
y = 22;
...
```

Namensräume

2. Eigenschaften von Namensräumen

Beispiele für explizite Namensräume

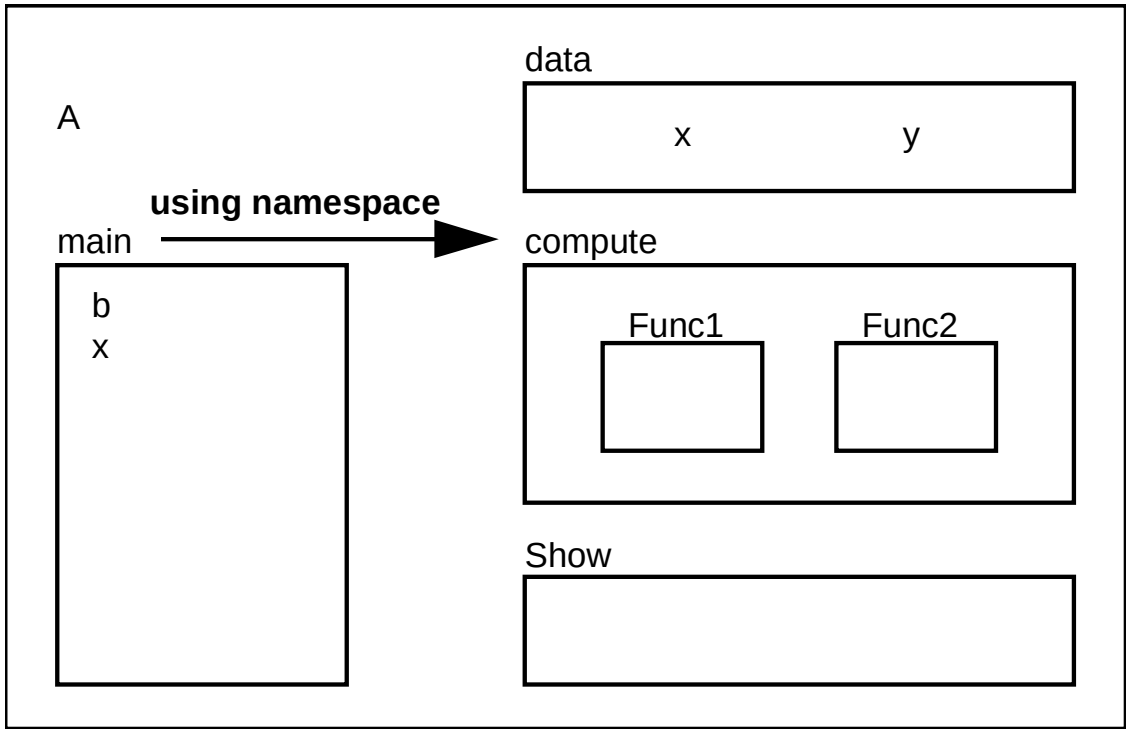
```
...
// Uses a global namespace, local namespaces in functions,
// a namespace std and two explicitly defined namespaces
// data and compute

const int A = 20; // global constant

namespace data
{
  int x, y;
}

namespace compute
{
  int Func1(int x, int z)
  {
    return x + z + A;
  }

  int Func2( int y, int z)
  {
    return y - z - data::x;
  }
}

void Show(std::string s,int z)
{
  std::cout << s << z << std::endl;
}

int main(void)
{
  using namespace compute
  const int B = 10;
  int x;
  x = Func1(::A, B);
  data::y = Func2(B, data::x);
  Show("x = ", x);
  ::Show("x = ", data::x);
  Show("y = ", data::y);
  return 0;
}
```

B1 B2 B21 B22 B3 B4

Namensräume

2. Eigenschaften von Namensräumen

Namensraummodell für das Beispiel mit expliziten Namensräumen



Namensräume

2. Eigenschaften von Namensräumen

Komposition von Namensräumen

```
namespace  A {
  int x = 1;
  int y = 2;
}

namespace B {
  int a = 3;
  int b = 4;
}

namespace C {
  using A::x;
  using A::y;
  using namespace B;
}

namespace D {
  using namespace C;
}

…
```

Zugriffsbeispiele

```
…
std::cout << A::x << std::endl; // 1
std::cout << A::y << std::endl; // 2
std::cout << B::a << std::endl; // 3
std::cout << B::b << std::endl; // 4
std::cout << C::x << std::endl; // 1
std::cout << C::y << std::endl; // 2
std::cout << C::a << std::endl; // 3
std::cout << C::b << std::endl; // 4
std::cout << D::x << std::endl; // 1
std::cout << D::y << std::endl; // 2
std::cout << D::a << std::endl; // 3
std::cout << D::b << std::endl; // 4
…
```

Namensräume

2. Eigenschaften von Namensräumen

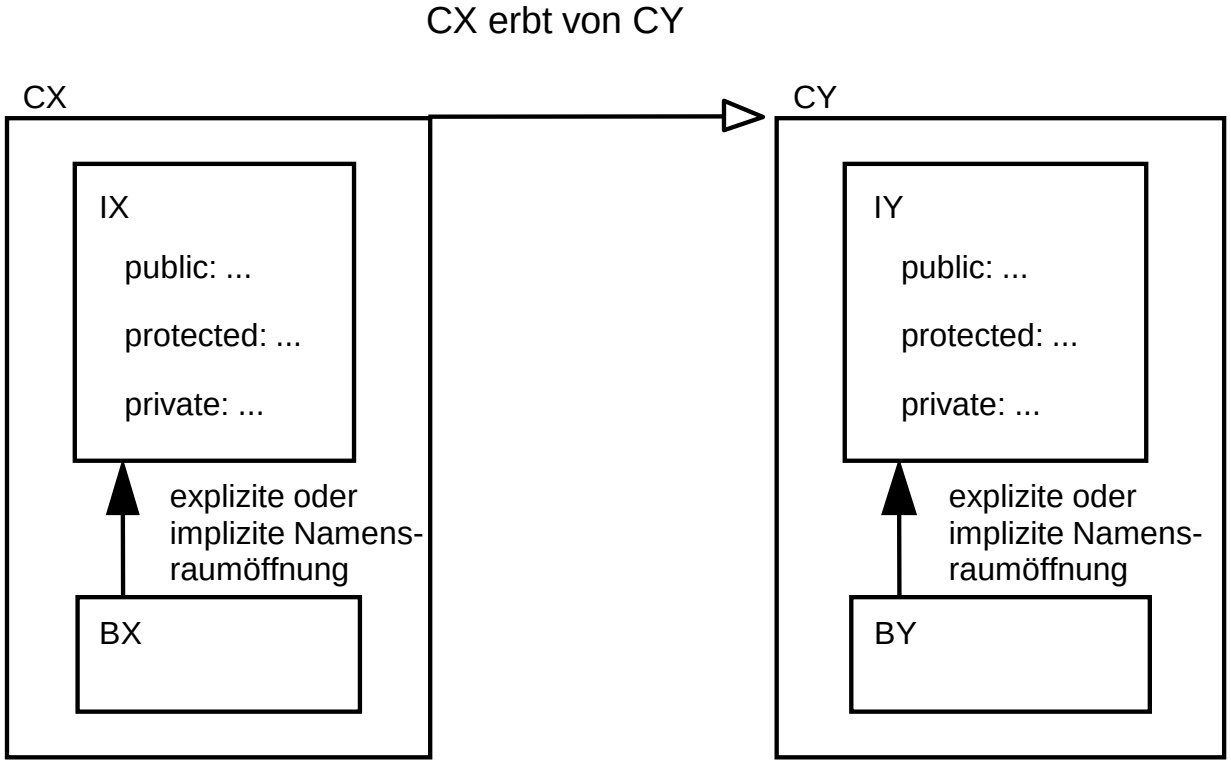
Namensräume von Softwarebauelementen

- Softwarebauelemente für die Programmierung im Großen sind separate Fertigungseinheiten. Sie verfügen über eine Schnittstelle (Interface), die beschreibt, welche Funktionalität das Bauelement nach außen zur Verfügung stellt.
- Die Bauelementeschnittstelle bildet einen Namensraum.
- Die Realisierung der Funktionalität des Softwarebauelements erfolgt in einem Implementationsteil (Body). Der Implementationsteil des Softwarebauelements ist privat und für andere Softwarebauelemente nicht zugänglich. Im allgemeinen Fall ist er ebenso wie der Interfaceteil eines Softwarebauelements eine Fertigungseinheit.
- Der Body eines Softwarebauelements besitzt einen eigenen Namensraum. Der Namensraum des Softwarebauelementeinterfaces muss explizit oder implizit für den Implementationsteil geöffnet werden.
- Sowohl Bauelementinterface als auch Bauelementbody können die Öffnung von Namensräumen anderer, zur Erfüllung der Funktionalität des Bauelements benötigter Softwarebauelemente, anfordern. Die Namensräume werden geöffnet, wenn Softwarebauelemente Beziehungen zueinander haben.
- Durch die Namensraumöffnung werden für Softwarebauelemente resultierende Namensräume gebildet, in denen ein Zugriff erfolgen kann.

Namensräume

2. Eigenschaften von Namensräumen

Namensraummodell für Softwarebauelementestrukturen



CX, CY - Klassen als Softwarebauelemente

IX: Namensraum des Interface von CX

IY: Namensraum des Interface von CY

BX: Namensraum der Implementation (Body) von CX

BY: Namensraum der Implementation (Body) von CY

Namensräume

3. Objekträume und Speichermodelle

Glossar

Lebenszyklus eines Objekts

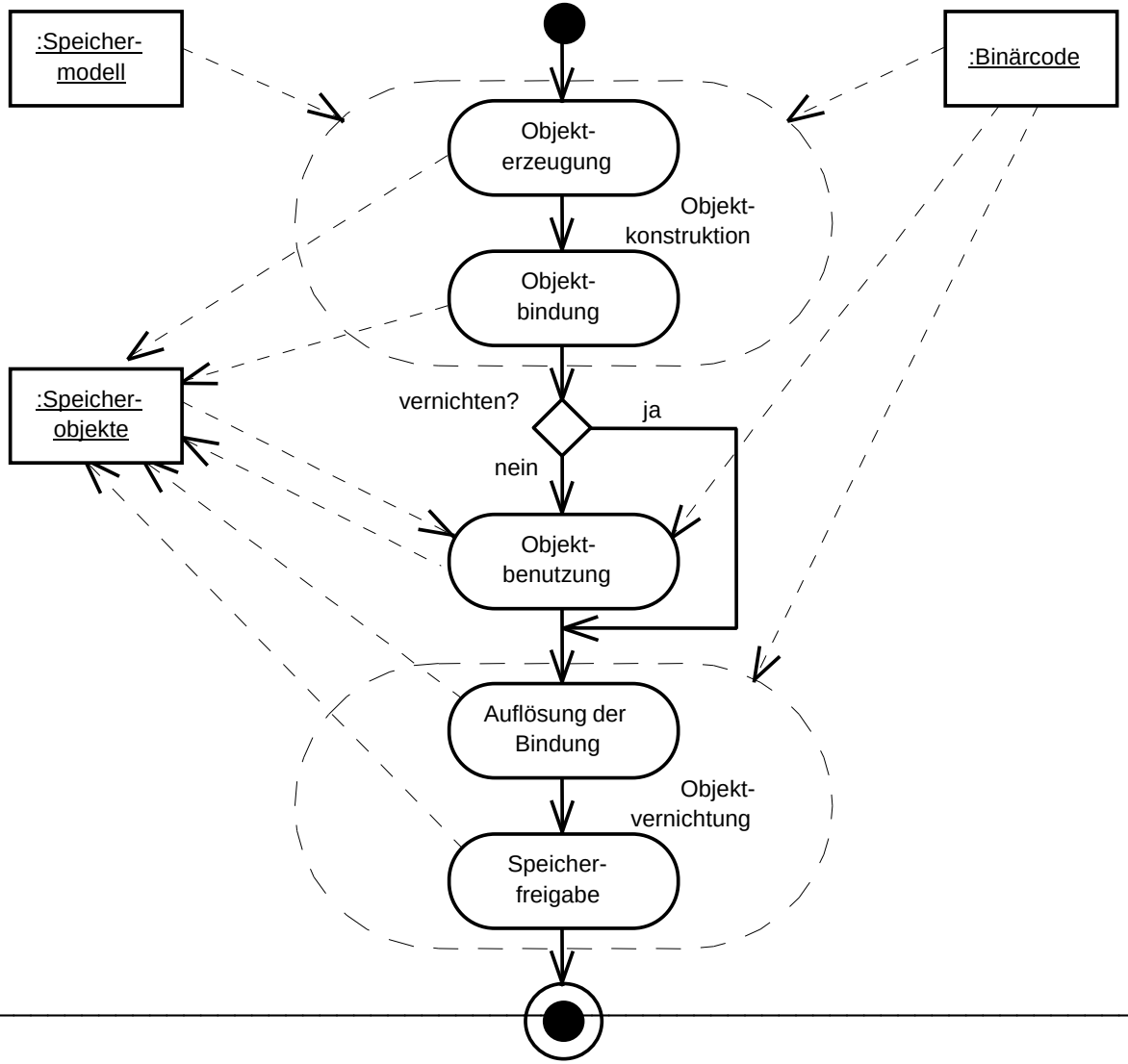
- Ein Speicherobjekt (Objekt) ist an einen Speicherplatz , der durch eine (virtuelle) Maschine aus einer Deklaration eines programmtechnischen Objekts erzeugt wurde und der einen Identifikator besitzt gebunden. Der Identifikator ist, falls die Maschine über einem Adressraum arbeitet, eine Adresse. Ein Name oder mehrere Namen können an ein Objekt gebunden werden.

- Ein Objekt hat einen Lebenszyklus.

Namensräume

3. Objekträume und Speichermodelle

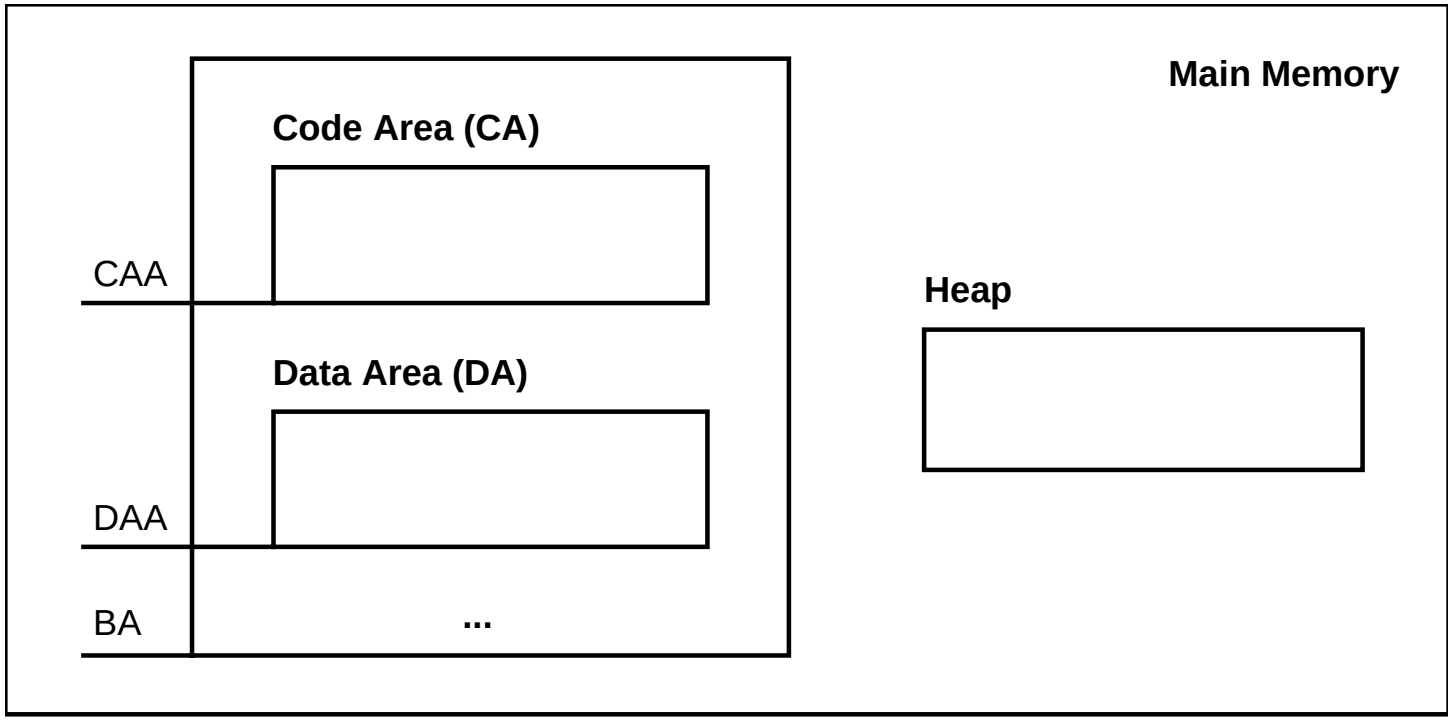
Lebenszyklusmodell eines Objekts (Verfeinerung der Binärcodeverarbeitung)



Namensräume

3. Objekträume und Speichermodelle

Beispiel für ein Speichermodell



BA: Base Address DAA: Data Area Address
CAA: Code Area Address

Namensräume

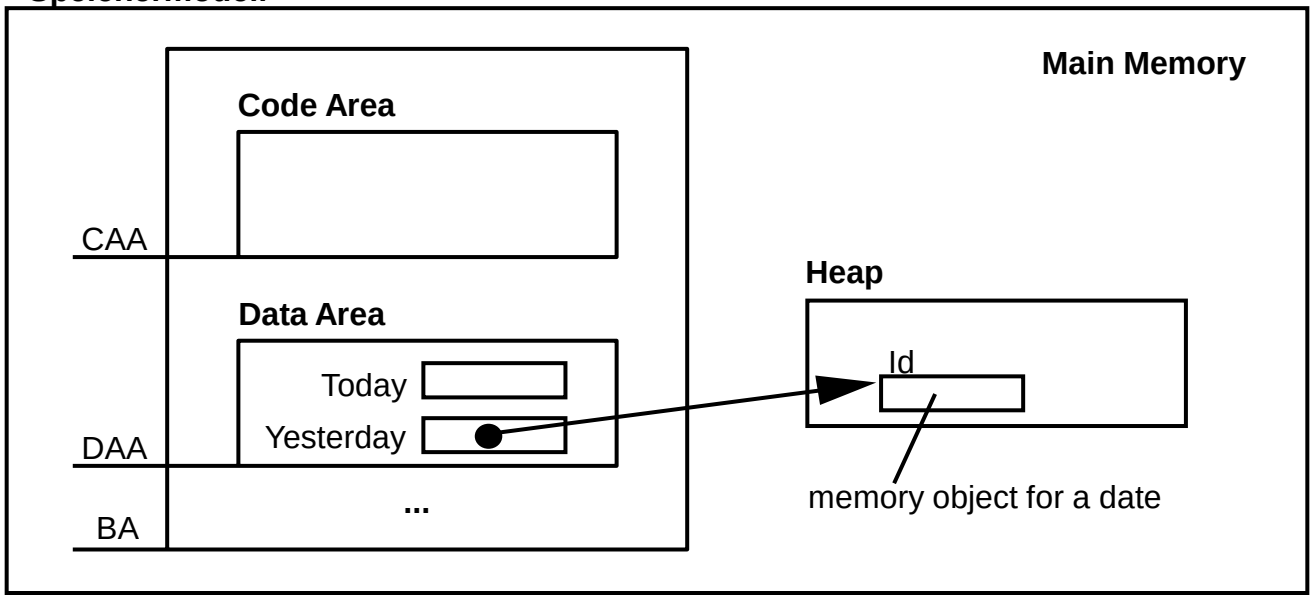
3. Objekträume und Speichermodelle

Beispiele für Statische und dynamische Objekterzeugung

Programm

```
...
TDate  Today;
TDate *Yesterday;       //  allocates memory for a pointer variable
Yesterday = new(TDate); //  allocates memory for an object of
                        //  type TDate and binds this memory to
                        //  pointer variable
...
```

Speichermodell



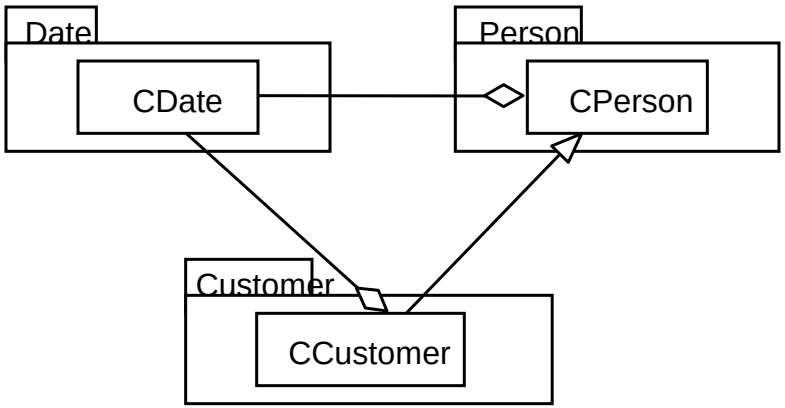
Namensräume

4. Allgemeine Beschreibungssprache für Softwarebauelemente Construction Element Definition Language (CEDL)

- Ziel der Sprache ist die Bereitstellung programmiersprachenunabhängiger Ausdrucksmittel für die Beschreibung von Softwarebauelementen.
- Die Sprache soll für alle Typen von Softwarebauelementen sowie zur Interpretation der Architektur-, Namensraum- und Verarbeitungsmodelle geeignet sein.
- Ziel ist nicht die formale Beschreibung von Architekturen oder etwa die Erzeugung von ablauffähigen Programmen aus den Beschreibungen.

Beispiel

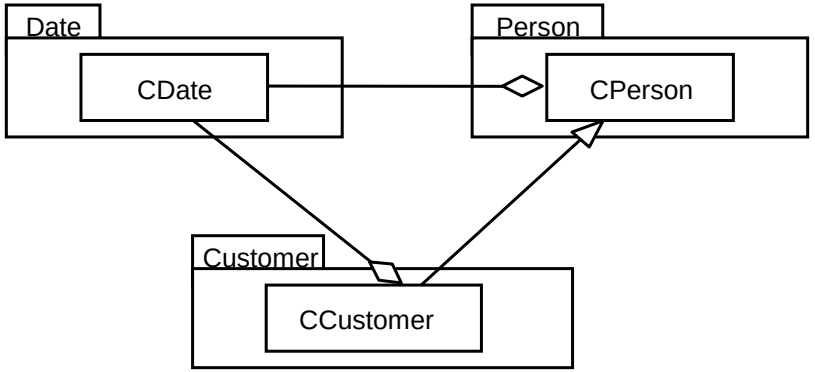
Ausschnitt einer Klassenstruktur



Namensräume

4. Allgemeine Beschreibungssprache für Softwarebauelemente

CEDL Beschreibung

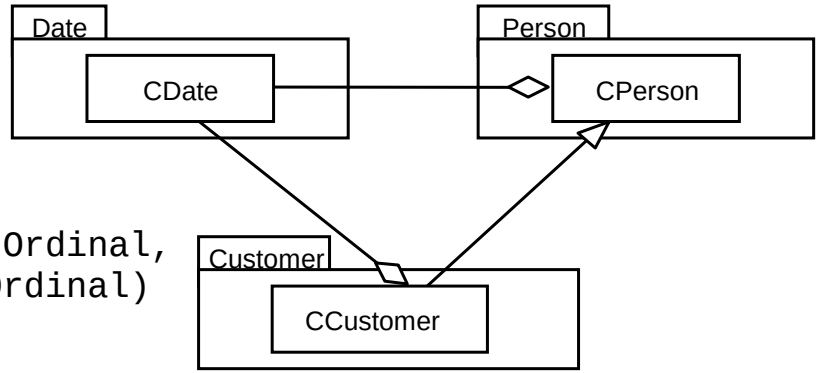



```
unit Date

  interface

    class CDate
      attributes
        Day : Ordinal,
        Month : Ordinal,
        Year : Ordinal;

      operations
        InitDate(in da: Ordinal, in mo: Ordinal,
                 in ye: Ordinal),
        // constructs and initializes a date
        virtual Free(),
        // deconstructs a date
        ...
        Equal(in dat: CDate, out res: Boolean);
        // compares two dates

    classend

  interfaceend
```

Namensräume

4. Allgemeine Beschreibungssprache für Softwarebauelemente

CEDL Beschreibung




```
Implementation

  variables Date1: CDate;

  // Class Implementation
  operation CDate.InitDate(in da: Ordinal,
          in mo: Ordinal, in ye: Ordinal)
    ...
    body
      ...
    bodyend
  operationend,

  operation CDate.Free()
    ...
  operationend

  ...

  operation CDate.Equal(in dat: CDate, out res: Boolean)
    ...
  operationend

implementationend
...
unitend
```

Namensräume

5. Verarbeitungsmodelle

Begriff

- Verarbeitungsmodelle erklären auf der Grundlage der Namensraum- und Objektlebenszyklusmechanismen für Softwarebauelemente wie das Zusammenwirken von Bauelementen eines Softwareprodukts erfolgt.

- Zum einen hat das Verarbeitungsmodell enge Bezüge zur Bindung von Namen an Speicherplätze, insbesondere zu den Bindungsarten Zeitpunkt und Ort, zum anderen gibt es zwei Grundmechanismen der Abarbeitung von Programmen, die compilative und die interpretative Verarbeitung.

- Bei der compilativen Verarbeitung wird ein in einer Programmiersprache geschriebenes Programm vor der Abarbeitung in ein abarbeitbares Programm übersetzt.

- Bei der interpretativen Verarbeitung werden die Anweisungen eines in einer Programmiersprache vorliegenden Programms nach den notwendigen syntaktischen Überprüfungen sofort ausgeführt.

- Verarbeitungsmodelle sind dynamisch, da sie die Prozesse der Codeverarbeitung zeigen Sie können entweder durch Folgen von Zustandsänderungen (von innen) oder durch Animation (von außen) dargestellt werden.

Namensräume

6. Beispiele für Modultypen in C/C++

Das Modul als Softwarebauelementetyp

- Ein Modul ist ein komposites Softwarebauelement, das aus elementaren Softwarebauelementen und Moduln bestehen kann.
- Ein Modul realisiert eine prozedurale Abstraktion und/oder eine eine Datenabstraktion und stellt Leistungen für andere Module an einer Schnittstelle bereit.
- Ein Modul besteht aus einem Modulinterface und einer Modulimplementatation.
  Das Modulinterface spezifiziert Typen, Daten und Operationen, die einem zusammengehörigen Problembereich zuzuordnen sind und für die Benutzung durch andere Module bereitgestellt werden.
- Module sind in Units (Teilsysteme) eingebettet.
  Das Modulinterface wird im Interface der Unit deklariert.
  Die Unitimplementation beinhaltet die Modulimplementation.
  Mehrere Module können in eine Unit eingebettet sein.
  Das Interface oder Teile von diesem sowie die Implementation eines Moduls oder Teile der Implementation können in verschiedenen Units enthalten sein.
- Beim Import einer Unit werden die Namen aller in der Unit deklarierten Module bekannt gemacht.
  Ein qualifizierter Import von Modulen aus Units ist möglich.
  Eine Öffnung der Namensräume der importierten Module erfolgt durch die Import-Beziehung zwischen Units nicht.

Namensräume

6. Beispiele für Modultypen in C/C++

Datenkapsel

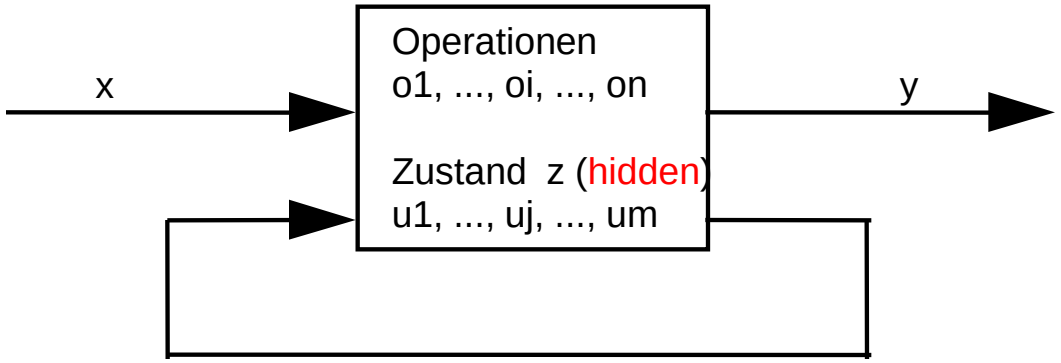
- Sie ist hat spezielle Moduleigenschaften
- Sie verfügt über eine Menge aufrufbarer, an der Schnittstelle sichtbarer Operationen.
- Der Benutzer der Datenkapsel muss nicht wissen, wie deren Datenobjekte und Operationen realisiert sind. Es können auch Operationen existieren, die an der Schnittstelle des Moduls nicht sichtbar sind (interne Operationen).
- Der Benutzer kann auf die Datenobjekte nicht direkt, sondern nur über Operationen der Modulschnittstelle zugreifen (Zugriffsoperationen).

Namensräume

6. Beispiele für Modultypen in C/C++

Datenkapsel

Modell der Datenkapsel



Operationen o1, ..., oi, ... , on
oi: X x Z → Y

Zustandsübergänge u1, ..., uj, ..., um
uj: X x Z → Z

Namensräume

6. Beispiele für Modultypen in C/C++

Datenkapsel

Es können zwei **Arten von Datenkapseln** unterschieden werden:

- Abstrakte Datenstruktur (ADS) und
- Abstrakter Datentyp (ADT).

Eine Abstrakte Datenstruktur verkapselt eine Anzahl von statisch deklarierten Variablen, im Grenzfall ist dies eine Variable.
Die ADS ist eine statische Realisierung der Datenkapsel.

Der Abstrakte Datentyp gestattet die Erzeugung von Variablen des Typs zur Laufzeit und ist eine dynamische Realisierung der Datenkapsel.

Namensräume

6. Beispiele für Modultypen in C/C++

Abstrakte Datentypen in C++

- Im Modulinterface wird der Typname als Zeiger vom Typ `void*` deklariert. Der Typname hat entsprechend unserer Namenskonventionen den Präfix `U` für „untypisiert“.
- In der Modulimplementation wird der geheim gehaltene Typ (englisch: hidden type) deklariert.
- In der Implementation der Schnittstellenfunktionen ist jeweils eine Typanpassung (englisch: casting) erforderlich, da für die Prozedurparameter vom Typ `void*` in den Funktionen die jeweiligen typisierten Zeiger verwendet werden müssen.

Namensräume

6. Beispiele für Modultypen in C/C++

Abstrakte Datentypen in C++

Modulinterface des Abstrakten Datentyps `MPersonADT`

```
...
typedef struct {//!!!
  unsigned int Day;
  unsigned int Month;
  unsigned int Year;
} TDate;

enum TSex {m, f};

namespace MPersonADT {
  typedef void* UPerson;

  void ConPerson(UPerson* pers);  // constructs a UPerson
  void Free(UPerson pers);        // destroys a UPerson
  void InitPerson(UPerson pers, const char* fn, const char* na,
          const char* ad, TDate dbd, TSex se);
  bool Equal(UPerson pers1, UPerson pers2);
  ...
};
...
```

Namensräume

6. Beispiele für Modultypen in C/C++

Abstrakte Datentypen in C++

Implementation des Abstrakten Datentyps `MPersonADT`

```
...
#include "PersonADT.h"

typedef struct {   // declaration of the hidden type
  char* FirstName;
  char* Name;
  char* Address;
  TDate Birthday;
  TSex  Sex;
} TPersonIntern;
```

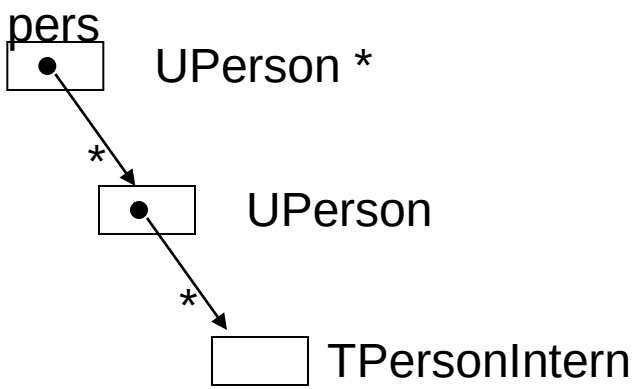
Namensräume

6. Beispiele für Modultypen in C/C++

Abstrakte Datentypen in C++

Implementation des Abstrakten Datentyps `MPersonADT`

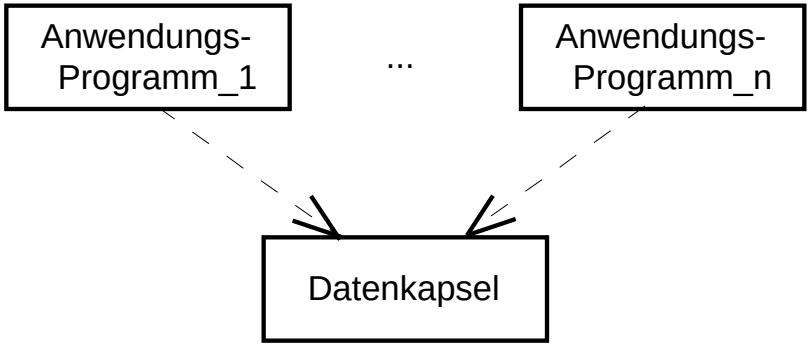
```
void MPersonADT::ConPerson(UPerson *pers) {
  *pers = (UPerson) malloc(sizeof(TPersonIntern)); // new
  ((TPersonIntern*) *pers) -> FirstName = (char*) malloc(strlen("*") + 1);
  strcpy(((TPersonIntern*) *pers) -> FirstName, "*");
  ...
  ((TPersonIntern*) *pers) -> Birthday.Day = 1; // information hiding!
  ...
  ((TPersonIntern*) *pers) -> Sex = f;
}

void MPersonADT::Free(UPerson pers) {
  free(((TPersonIntern*) pers) -> FirstName);
  free(((TPersonIntern*) pers) -> Name);
  free(((TPersonIntern*) pers) -> Address);
  free(((TPersonIntern*) pers));
  pers = NULL;
}
```



Namensräume

6. Beispiele für Modultypen in C/C++

Applikationsschema bei Nutzung einer Datenkapsel



Namensräume

6. Beispiele für Modultypen in C/C++

Diskussion der Eigenschaften der Modultypen

Abstrakte Datentypen

Vorteil:

Typen werden geheim halten.

Nachteile:

- Programmierung erfolgt auf einer sehr technischen Ebene (Pointer, casting),
- Erhöhung der Gefahr von Programmierfehlern,
- Rechenzeiten verlängern sich.