4 Module, Datenabstraktion (Programmstrukturierung ohne Objektorientierung)

Lernziele:

Module als Bausteine der Architektur
Datenabstraktionsprinzip
Datenobjektmodule, Datentypmodule (Klassen)
Simulation von Modulen in C++

Module und Architektur

Module als Bausteine

– bisher Hauptprogramm und Prozeduren

– Prozedur: ein Dienst an der Schnittstelle
  Module: mehrere Dienste, die “zusammengehören”

– große Programme: Aufteilung in Module (und Teilsysteme)
  Gründe: Entwurf vor Realisierung
  Überprüfung gegen Problemstellung vor Programmerstellung
  Überprüfung Programmerstellung gegen Entwurfsspezifikation
  Wartbarkeit (Anpaßbarkeit, Portierbarkeit)
  Arbeitsteilung
  Qualitätssicherung
  Dokumentation
  etc.

– Export: Was der Modul für andere zur Verfügung stellt
  Import: Was der Modul zu seiner Realisierung an Hilfe braucht
  bisher z. B. Import von `iostream.h`

– Module eröffnen
  - Erstellung von Programmbausteinen und Ablage in Programmbibliothek
  - getrennte Bearbeitung und Übersetzung

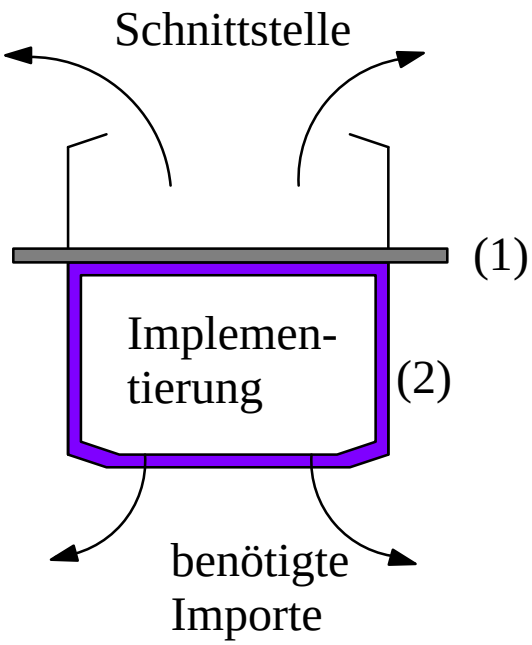
Module und Abstraktion

– Modul ist Abstraktion (1)

von Implementierungsdetails des Moduls

von hierzu benötigten Importen

Bsp.: `iostream.h`,
wie bisher verwendet



– Modul ist Kapsel (Information Hiding) (2)

verbirgt Interna

Programmiersprache verhindert ggfs. (nicht C++)
Zugriff auf diese: Sie sind nicht sichtbar.

– Module

Definitionsteil (Schnittstelle)
–> Export, öffentlich zugänglich

Implementationsteil (Rumpf des Moduls)
–> verborgen

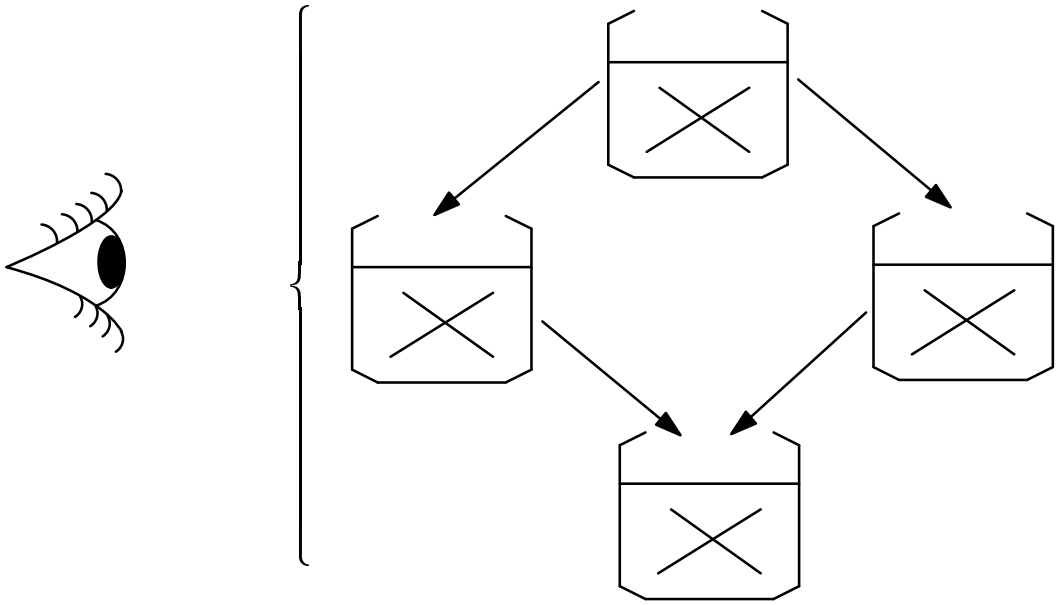
beide Teile sind getrennt übersetzbar

benötigen weitere Dienste (Importe)

Module in der Architektur

– Modul ist Teil einer Softwarearchitektur
  (→ Vorlesung Softwaretechnik, insb. Architekturmodellierung)
  diese erhält Struktur durch
    Verwendung bestimmter Arten von Modulen
    Verwendung bestimmter Arten von Importen
    Konsistenzbedingungen
    ist Hierarchie bestehend aus vielen Bausteinen

– besondere Art der Abstraktion



alle Bausteine aufgeführt,
insb. auch in tiefen Schichten,
Details der Bausteine (Rümpfe) nicht betrachtet
idealisierte Vorstellung;
geht praktisch nicht anders

Schnittstelle und Rumpf

Schnittstellen von Modulen

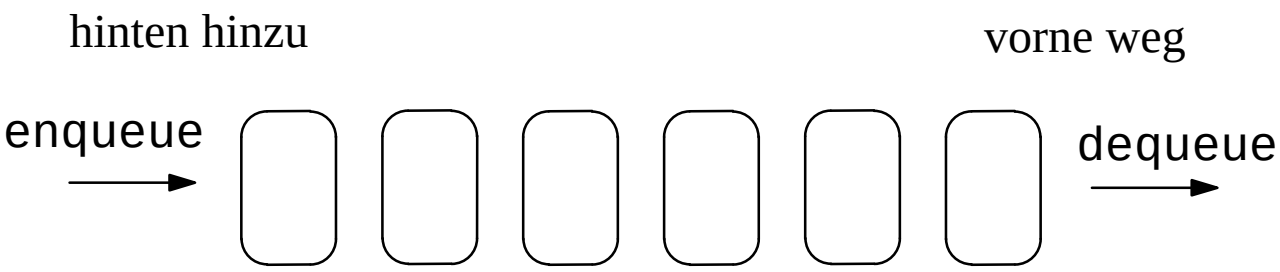
– Schnittstelle enthält exportierte Objekte
(Ressourcen, die Modul anderen zur Verfügung stellt)
nämlich:

Konstante	(Vorsicht → Datenabstraktion)
Variable	(Vorsicht → Datenabstraktion)
Typen	(Vorsicht → Datenabstraktion)
Prozeduren	(nur Schnittstellen, Rümpfe im Implementationsteil)

– Gültigkeit/Sichtbarkeit in allen importierten Modulen
Konstante
Variable mit Feinstruktur } Vorsicht:
Typen mit Feinstruktur } Veränderung
(bei transparenten (offenen) Typen)
Prozeduren (Schnittstelle, aber nicht Rumpf)

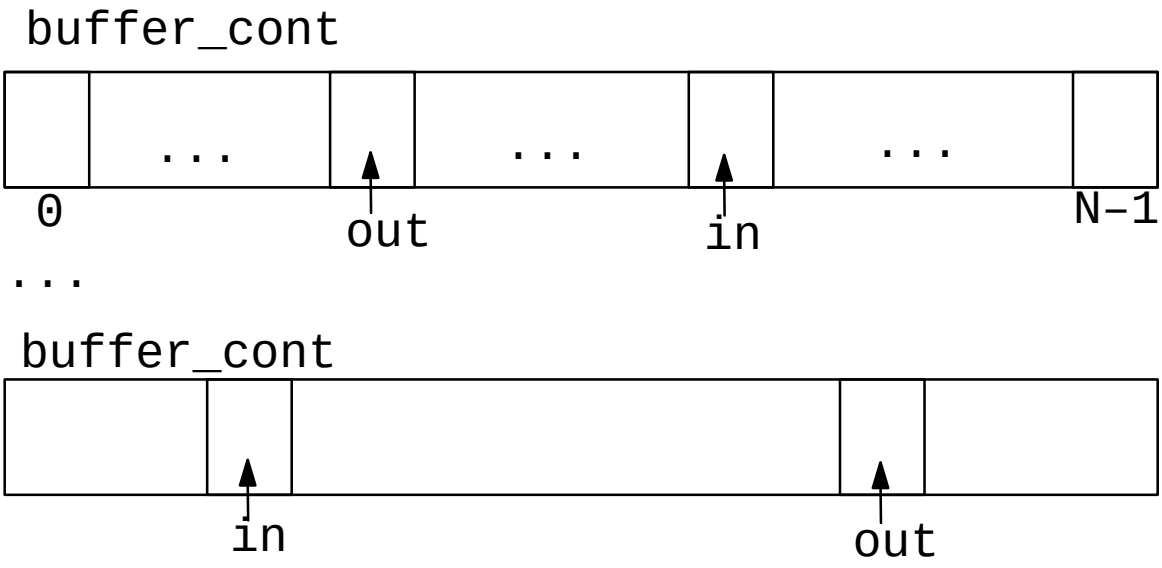
– Typen
transparente (s.o.)
opake: Beschränkung auf Zeiger
Details im Rumpf gekapselt

– Schnittstelle sollte logisches Protokoll definieren,
z.B. Puffer (buffer, queue)

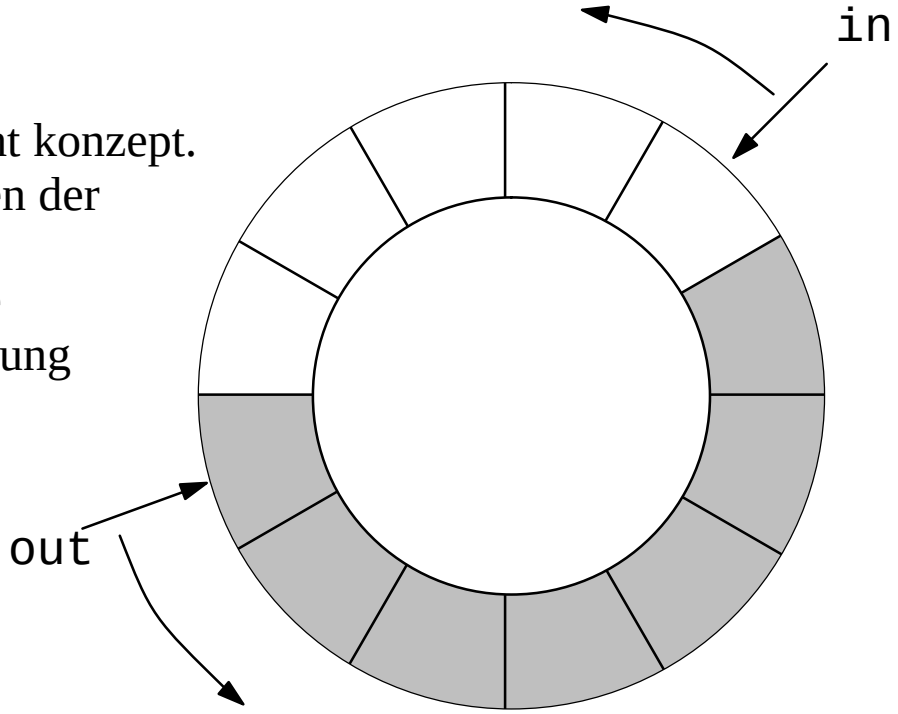


Zugriffsstrategie FIFO

– Rumpf (Implementation) enthält Realisierung der Schnittstelle
  - Realisierung z.B. zirkuläre Speicherung



entspricht konzept. Verkleben der Enden: zirkuläre Speicherung



```
// Schnittstelle (interface, protocol)=====

void enqueue(item x);
void dequeue(item &x);
bool nonEmpty();
bool nonFull();
// Sei item ein Datentyp fuer die Elemente
// des Puffers (austauschbar).
// enqueue fuegt hinten ein Element dazu
// dequeue nimmt vorne ein Element weg und
// liefert dabei dessen Wert.
// Voraussetzung von enqueue:
// Datenstruktur hat noch Platz;
// fuer dequeue: Datenstruktur nicht leer.
// ---------------------------------------
```

```
// Rumpf (body, implementation) ------------
// Datenstruktur mit zirkulaerer Speicherung
// Realisierung der Schnittstelle ist darauf
// abgestimmt

unsigned int const N=100; //Behaeltergroesse
static int In=0;  // Werte: 0,...,N-1;
static int Out=0; // Verwaltung der Enden
static int n=0;   // Wert: 0,..,N; Anzahl
                  // der Elemente
typedef itemT it_BT[N];
static it_BT bc;  // Behaelter fuer Buffer
static bool nempty=false, nfull=true;

void enqueue (itemT x) {
   if (n<N) {
      bc[In]=x; In=(In+1)%N;
      n++; nfull=(n<N); nempty=true;
   }
}

void dequeue (itemT &x) {
   if (n>0) {
      x=bc[Out]; Out=(Out+1)%N;
      n--; nempty=(n>0); nfull=true;
   }
}

bool nonEmpty() {
   return nempty;
}

bool nonFull() {
   return nfull;
}
// ========================================
```

– andere Realisierung (z.B. verkettete Liste)
  ohne Auswirkung auf alle importierenden Module

– Information Hiding
  Realisierung:
    1) andere interne Datenstrukturen
    2) andere Implementation der Operationen
    3) in der Regel andere importierte Module

– Vorbedingung für Benutzung durch Klienten
  `nonEmpty` für `dequeue`
  `nonFull` für `enqueue`

Datenabstraktionsprinzip, Datenobjektmodule

Datenabstraktionsprinzip

– Datenstruktur mit Zugriffsoperation eine Einheit: Modul

– Zugriffsoperation nach außen:
  Funktionale (logische) Sicht der Schnittstelle:
    Zugriffsoperationen (Prototypen)

– Informationsverbergung auf "logischer" Ebene
  Realisierung der Datenstruktur
  Realisierung der Zugriffsoperation
  } verborgen
  } vgl. (1) von Abb. 1
    benötigte Importe

– Datenstruktur mit dieser Eigenschaft:
    abstraktes Datenobjekt
  Modul, der eine solche Datenstruktur realisiert:
    abstraktes Datenobjektmodul

– Bei Datenabstraktion sind stets Datenstrukturen mit Gedächtnis im Spiel

– keine Auswirkung auf Klienten bei Änderung der Realisierungstechnik

– Bsp. `buffer`
  Schnittstelle (unverändert)

  Realisierungsentscheidung
    Datenstrukturen
    Importe
    $\Rightarrow$ Realisierungstechniken für Zugriffsoperationen

– Datenabstraktion insbesondere bei Verwendung von Zeigern:
  Gefahr lokalisieren

– Datenabstraktion macht Schnittstelle zwischen Moduln “dünn”
  Überlegung:
    globaler Speicher für Puffer
    Pufferoperationen direkt realisiert an allen Stellen der Verwendung

– jede komplexe “Datenstruktur” ist in der Systemarchitektur zu sehen: jeweils Modul dafür

– Realisierungsbandbreite:
  auf Laufzeitkeller bei lokalem Modul
  auf statischem Speicher bei Bibliothekseinheit
  auf der Halde bei Verwendung von Zeigern
  auf Sekundärspeicher
  auf Knoten eines Netzes von Rechnern

– Schnittstellengestaltung von DA–Bausteinen
  Sicherheitsfragen (s.o.)
  return–Parameter (Zustandsparameter für Berechnung)

Datenobjektmodule

– bisher: `buffer` ist abstraktes Datenobjekt (Datenkapsel)
  abstrakt: über sauberes Protokoll mit Zugriffsoperationen gehandhabt

– abstr. Datenobjekt hat Gedächtnis (Zustand)
  wird bei Zugriffsoperationen `enqueue`, `dequeue` verändert;
  `nonFull` und `nonEmpty` fragen Zustand ab

– Verwendung (Datenobjekt über Namen der Op. identifiziert)

```
if (nonFull()) {
   enqueue(it);
} else { Error(...);
}
```

} Verwendung des abstr. Protokolls; Realisierung nicht zu sehen

– abstr. Datenobjekt ist Baustein der Architektur

– Verwendung für Grunddatenstrukturen, die nur einmal vorhanden sind:
  Datenbestände/Datentöpfe in sauberer Form

Hilfsmittel zur getrennten Übersetzung

Dateien für Schnittstellen und Rümpfe

– Unabhängige Übersetzung:
  Nicht das ganze Programm, sondern Teile hiervon werden einzeln geprüft.
  Konsistenz zwischen den Teilen (unzureichend) durch Binder

– Somit unabhängige Übersetzung:
  Separate Übersetzung ohne (mit teilweiser) Querprüfung
  Gewünscht: getrennte Übersetzung:
    separate Übersetzung mit Querprüfung
  separat übersetzte Einheiten = Dateien in C++

– separate Übersetzung wegen getrennter Bearbeitung und zur Verminderung der Recompilationszeit

– Header-Files
  Schnittstellenbeschreibungen von Funktionen,
  Modulen (zur Simulation von Modulschnittstellen)
  (Dateiname z.B. `a.h`)

– Implementation-Files
  enthalten Rümpfe bei Modulen, Funktionsdeklarationen bei Funktionen
  (zur Simulation von Modul–Rümpfen)
  (Dateiname z.B. `a.cpp`)

– Main-Files
  für das Hauptprogramm (Dateiname z.B. `mainprog.cpp`)

– Bibliotheks-Files
  enthalten Zusammenfassung vordefinierter Bausteine
  (zur Simulation von Teilsystemen)
  (Dateiname z.B. `stdlib.h`)

1) `#include`

Verwendung aber Neuübersetzung

```
#include ”a.cpp”
#include ”b.cpp”
void main() {
   proc_a1();  // Funktionsaufruf
   proc_a2();
   return func_b();
}
```

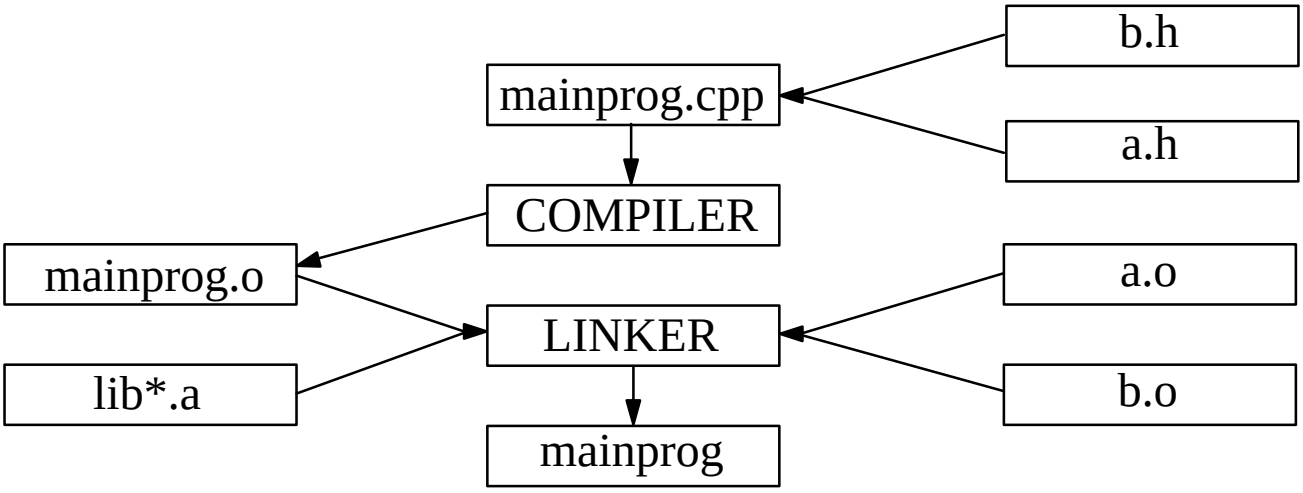
```
// a.h
void proc_a1();
void proc_a2();
```

```
// a.cpp
#include ”a.h”
void proc_a1() {
   // Programmcode zu proc_a1
}
void proc_a2() {
   // Programmcode zu proc_a2
}
```

```
// b.h
int func_b();
```

```
// b.cpp
#include “b.h”
int func_b() {
   // Programmcode zu func_b
}
```

```
// mainprog.cpp
#include ”a.h”
#include ”b.h”
void main() {
   proc_a1();
   proc_a2();
   return func_b();
}
```



Durch “Makefiles” oder “Projekte” werden größere Softwarevorhaben handhabbar.

2) `#ifndef, #define`

`#include "a.h"` kann zu Problemen führen, wenn die in `a.h` enthaltene Schnittstelle an mehreren Stellen benötigt und eingefügt wird.

```
// c.h
#ifndef __INC_C_H__
#define __INC_C_H__

void proc_c();
int const C_MAX = 100;

#endif
```

Nutzen dies später bei allen Schnittstellenangaben, um für Importe geeignetes Schema zu finden.

`extern` vor Deklaration: ist in anderer Datei festgelegt.
`static` vor Deklaration: Gültigkeit auf aktuelle Datei beschränkt.

Alle anderen Deklarationen (`auto`) in Prozeduren/Funktionen:
Variablendeklarationen auf Laufzeitkeller angelegt.

Funktionale Module

Allgemeines

– aktivitätsorientierte Module
  EA Verhalten
  kein Gedächtnis

– für Transformation (z.B. Compiler)
  für komplexe Berechnung (z.B. numerische Integration)

– Anzahl der Dienste:
  ein Dienst: Prozedur als Spezialfall
  mehrere Dienste: funkt. Modul, Funktionsmodul

– Welche Dienste zusammenfassen:
  gleiches Parametertypprofil
  evtl. zusätzlich ähnliche Realisierung

Beispiel in C++

– Schnittstelle

```
// function module Trigonometry ===========

#ifndef __INC_TRIGONOMETRY_H__
#define __INC_TRIGONOMETRY_H__

float acos(float x); // Arcus Cosinus
float asin(float x); // Arcus Sinus
float atan(float x); // Arcus Tangens
float cos(float x);  // Cosinus
float cosh(float x); // Cosinus Hyperbolicus
float sin(float x);  // Sinus
float sinh(float x); // Sinus Hyperbolicus
float tan(float X);  // Tangens
float tanh(float X); // Tangens Hyperbolicus
#endif
// end interface Trigonometry---------------
```

– Rumpf

```
// body Trigonometry------------------------

#include ”trigonometry.h” // Uebereinstimmung
                          // des Rumpfes mit
float acos(float x) {     // der Schnittstel-
   ...                    // le muss der Pro-
}                         // grammierer
float asin(float x) {     // gewaehrleisten;
   ...
}
  .
  .
  .
// end body Trigonometry=====================
```

Handhabung von Importen

– Importe zur Kennzeichnung der Verwendung eines Teils der Schnittstelle eines anderen Moduls

– Bsp. Modul `Grafik`
in dessen Rumpf wird von `Trigonometry` `sin` und `cos` verwendet

```
// with Trigonometry import sin, cos;
#include ”trigonometry.h”

// body Grafik------------------------------
// Der Programmierer ist verantwortlich
// dafuer, dass nur sin und cos benutzt
// werden.

void rotate(float x, float y, float w,
             float &newx, float &newy) {
   // Berechnet die Koordinaten des Punktes
   // (x,y) nach einer Rotation um den
   // Ursprung um den Winkel w (Bogenmass)
   // in (newx, newy)

   newx = x*cos(w) – y*sin(w);
   newy = x*sin(w) + y*cos(w);
}

...

// end body Grafik==========================
```

Datentypmodule als Klassen

Motivation

– jetzige Realisierung: Modul ist Gedächtnis (Datenobjektmodul)
  Was tun, wenn mehrere Datenobjekte benötigt werden?
  Quelltextvervielfältigung?

– Aussehen normaler Datenobjektdeklaration:

```
typedef ElementT BT[N];
BT b_obj;              //BT fuer Buffer Type
```

über normale Typ-deklaration definiert

Struktur von `BT` und somit `b_obj` ist auf bel. Strukturniveau bekannt

gegen Datenabstraktionsidee!

– Wollen stattdessen:

```
...
if (nonfull(b_obj)) {  // wird spaeter in
   enqueue(b_obj, it1) // C++ anders notiert
   ...
} else {
   ...
}
```

– Brauchen Modul der
  "Typbezeichner"
  Zugriffsoperationen für Objekte dieses Typs } exportiert
  und sonst nichts: abstr. Datentypmodul

– kein Modul für Gedächtnis
  sondern für Schablone für Gedächtnisse

– Operationen haben einen “Parameter mehr”,
  für das entspr. abstrakte Datenobjekt

– Erzeugung der abstrakten Datenobjekte
  über Deklarationen
  über Erzeugungsoperationen
    “analog” zu `new` bei Haldenverwaltung

– Methodikregel:
    abstrakte Datenobjekte werden ausschließlich über
    Zugriffsoperationen verändert

– Schnittstelle

```
// ADT fuer Puffer ========================
#ifndef __INC_BUFFER_TYPE_H__
#define __INC_BUFFER_TYPE_H__

class Buffer_Type { // Interface
// Der Modulname ist auch der Typ, der
// ausserhalb verwendet wird; Objekte koennen
// damit deklariert und erzeugt werden.
// Handhabeung der Objekte nur ueber
// Zugriffsoperationen.
// Diese haben die folgende Semantik: ...

   // Protokoll fuer Verwendung ausserhalb
   // durch Klienten
public:
   Buffer_Type();
   void enqueue(item x);
   void dequeue(item &x);
   bool nonempty();
   bool nonfull();

private: // physische Schnittstelle f. Compiler
   static unsigned const N=10;
   int In;              // Dieser Teil ist
   int Out;             // aussen nicht zu
   unsigned n;          // verwenden.
   itemT bc[N];          // Er beschreibt den
   bool nempty;         // Aufbau der Objekte.
   bool nfull;
};
#endif Buffer_Type_h
// end interface Buffer_Type------------------
```

– Rumpf der Klasse

```
// body ADT Buffer_Type------------------

#include “Buffer_Type.h”

Buffer_Type::Buffer_Type() {
   // Erzeugung und Initialisierung
   // der Objekte
   In=0; Out=0; n=0;
   nempty=false; nfull=true;
}

void Buffer_Type::enqueue(itemT x) {
   if (n<N) {
      bc[In]=x; In=(In+1)%N; n++;
      nfull=(n<N); nempty=true;
   }
}

void Buffer_Type::dequeue(itemT &x) {
   if (n>0) {
      x=bc[Out]; Out=(Out+1)%N; n--;
      nempty=(n>0); nfull=true;
   }
}

bool Buffer_Type::nonempty() {
   return nempty;
}

bool Buffer_Type::nonfull () {
   return nfull;
}

// end body Buffer_Type=====================
```

```
//---Erzeuger---------------------
#include “Buffer_Type.h”

void erzeuge(Buffer_Type buffer) {
  ...
  if(buffer.nonfull()) {
    buffer.enqueue(daten);
  }
}

//---Verbraucher------------------
#include “Buffer_Type.h” // nonempty,dequeue

...


//---Kontrollmodul----------------
#include “Buffer_Type.h”  // Konstruktor
#include “Erzeuger.h”     // erzeuge
#include “Verbraucher.h”  // verbrauche

int main()
{
  Buffer_Type buffer;

  erzeuge(buffer);
  verbrauche(buffer);

  ...

  return 0;
}
```

Module und Simulation in C++

Simulation von Modulen

– funktionale Module
  abstr. Datenobjektmodule
  } Schnittstellen waren zu simulieren

  abstr. Datentypmodule: Schnittstellen durch Klassenkonstrukt zusammengefaßt

– Rümpfe aller dieser Modularten
  existieren nur als gedankliche Zusammenfassungen;
  in jeweils einer Implementationsdatei

– Schnittstellen → Header–Files
  Rümpfe → Implementation–Files
  damit Module simulierbar, wie oben gezeigt

– Importe simuliert durch `#include "M.h"`

– zur Vermeidung von Problemen beim mehrfachen Import

```
#ifndef __INC_M_H__
#define __INC_M_H__
...
#endif
```

Disziplin bei Simulation

– Durch `#include` wird die ganze Schnittstelle importiert;
daß nur die angegebenen verwendet werden: Disziplin

– Übereinstimmung zwischen Schnittstellen und Rumpf:
Disziplin
Schnittstelle muß realisiert werden,
dabei dürfen weitere lok. Hilfsmittel deklariert werden

– Abschottung des Rumpfes: Disziplin
Datenstrukturen bei ADOs dürfen außen nicht
verwendet werden,
wird durch `static` garantiert
bei ADTs (Klassen) wird Zugriff durch `private`
verhindert