Klassen und Objekte

Retrosektive

- Für die Modellierung von Dingen oder Sachverhalten nutzt man in c Strukturen.
- Strukturen sind dabei benutzerdefinierte Datentypen, die Komponenten verschiedenen Datentypes zu einem neuen Datentyp vereinen.
- Von diesem Datentyp können (beliebig) viele Objekte erzeugt werden, die dann jeweils ein Ding oder einen konkreten Sachverhalt modellieren.

- In diesem Sinn wird ein konkreter Student, z.B. Hans Huckebein, durch ein Objekt zugegebenermaßen abstrahiert beschrieben.
- Alle Objekte, die in dieser Weise Studenten beschreiben, werden ihrerseits durch den Dytentyp tStudent beschrieben.
- Die Deklaration eines solchen benutzerdefinierten Datentyps erfolgt in den meisten Fällen in einem Headerfile, in dem auch die Prototypen der Funktionen, die zur Arbeit mit eben diesem Datentyp bereitgestellt werden, angegeben sind.

Als Beispielm ein struct btime

```
#ifndef _H_BTIME_
#define _H_BTIME_
#include "btime.h"

typedef struct
{
  int h;
  int m;
}btime;

btime btimeGetCurrentTime();
void btimeShow(btime* pt);
btime btimeIncrement(btime t);

#endif
```

Datenstruktur

Funktionsprototypen dazu

- Die struktur btime dient der Modellierung einer Uhrzeit, bestehend aus Sunde(h) und Minute(m).
- Es gibt beispielhaft drei Funktionen zur Arbeit mit dem „Zeit-struct“.
- btime btimeGetCurrentTime();
  - Besorgt die aktuelle Systemzeit und trägt davon die Stunde und die Minute in eine neue Variable unserer Struktur und gibt diese als Wert zurück.
- void btimeShow(btime* pt);
  - Gibt die Werte als Zeit im Format hh:mm aus
- btime btimeIncrement(btime t);
  - Berechnet die Zeit der nächsten Minute

- Implementiert sind die drei Funktionen in der c-Datei btime.c wie folgt:

```
#include <stdio.h>
#include <time.h>
#include "btime.h"

btime btimeGetCurrentTime()
{
  btime tmp={};
  time_t lc = time(NULL);
  struct tm*plc = localtime(&lc);
  int h,m;
  tmp.h=plc->tm_hour;
  tmp.m=plc->tm_min;
  return tmp;
}
```

```
void btimeShow(btime* pt)
{
  printf("%02d:%02d",
                    pt->h,pt->m);
}

btime btimeIncrement(btime t)
{
  t.m++;
  if (t.m==60)
    {t.m = 0; t.h++;}
  if (t.h==24)
    {t.h = t.m = 0;}
  return t;
}
```

- Verwendung findet unser keiner btime – Modul in der c-Datei btimeMain.c

```
#include <stdio.h>
#include <time.h>
#include "btime.h"

int main(int argc, char*argv[])
{
  btime bt={};
  if (argc==3)
    {bt.h=atoi(argv[1]); bt.m=atoi(argv[2]);}
  else bt=btimeGetCurrentTime();
  printf("time is now  ");
  btimeShow(&bt);
  putchar('\n');
  bt=btimeIncrement(bt);
  printf("next time is ");
  btimeShow(&bt);
  putchar('\n');
}
```

- Das Ganze können wir nun mit

  gcc btime.c btimemain.c
- Oder

  gcc btime*.c
- Nach a.out compilieren und linken.

```
./a.out
time is now  18:04
next time is 18:05
beck@U330p:~/SS2016/cpp/class$ ./a.out 19 00
time is now  19:00
next time is 19:01
```

Zusammenfassend ist festzustellen:

- Es gibt einen Programmmodul btime bestehend aus
  - Einer öffentlichen Schnittstelle, die die Struktur und die dazugehörige Funktionalität in Form der Funktionen beschreibt (btime.h)
  - Einer Impelementationsdatei btime.c, die die Funktionalität implementiert. Dieser Teil könnte auch in Form einer vorkompilierten Datei btime.o oder als Library libbitime.a oder Teil einer Library vorliegen.
  - Die, im Headerfile enthaltene Struktur kann ggf. vor dem Benutzer auch noch verborgen werden, so wäre der Nutzer definitiv auf die öffentlichen Funktionen angewiesen, um mit dem Modul zu arbeiten.

Geändertes Beispiel mit verborgenem struct Typ

btimep.h

```
#ifndef _H_BTIME_
#define _H_BTIME_

struct btimes;
typedef struct btimes btime;

btime* btimeCreate();
void btimeSetH(btime*pbt, int h);
void btimeSetM(btime*pbt, int m);
void btimeGetCurrentTime(btime*);
void btimeShow(btime* pt);
void btimeIncrement(btime* t);

#endif
```

Leere struct-foreward deklaration
Allerdings kann der Benutzer
nun keine Variable, nur Pointer
des struct-Types anlegen!

btimep.c

```
#include <stdio.h>
#include <time.h>
#include <malloc.h>
#include "btimep.h"

typedef struct
{
  int h;
  int m;
}btime;

void btimeSetH(btime*pbt, int h)
   {pbt->h=h;}
void btimeSetM(btime*pbt, int m)
   {pbt->m=m;}

btime* btimeCreate()
{
  btime*pbt=malloc(sizeof (btime));
  pbt->h=pbt->m=0;
  return pbt;
}
```

Hier ist jetzt die vollständige Strukturdeklaration

```
void btimeGetCurrentTime(btime*bt)
{

  time_t lc = time(NULL);
  struct tm*plc = localtime(&lc);
  int h,m;
  bt->h=plc->tm_hour;
  bt->m=plc->tm_min;
}

void btimeShow(btime* pt)
{
  printf("%02d:%02d",pt->h,pt->m);
}

void btimeIncrement(btime* t)
{
  t->m++;
  if (t->m==60)
    {t->m = 0; t->h++;}
  if (t->h==24)
    {t->h = t->m = 0;}
}
```

```
#include <stdio.h>
#include <time.h>
#include "btimep.h"

int main(int argc, char*argv[])
{
  btime *pbt;
  pbt=btimeCreate();
  if (argc==3)
   {
      btimeSetH(pbt,atoi(argv[1]));
      btimeSetM(pbt,atoi(argv[2]));
   }
  else btimeGetCurrentTime(pbt);
  printf("time is now  ");
  btimeShow(pbt);
  putchar('\n');
  btimeIncrement(pbt);
  printf("next time is ");
  btimeShow(pbt);
  putchar('\n');
}
```

Vom c-struct zur Klasse

- Eine Klasse dient in gleicher Weise, wie ein struct-Datentyp der Modellierung.
- Betrachtet man die Funktionen zum c-Struct genauer, stellt man fest, dass alle einen Parameter von dem struct-Typ (Pointer) haben, mit dem sie die Operation durchführen sollen.
- Eine Klasse ist nun ein Verbund der Daten eines struct-Types von c und der Funktionen dazu.
- Da Funktionen und Daten nun eine Symbiose bilden, kann der erste Parameter auf magische Weise entfallen.

Schritte vom c-Struct zur Klasse

- Im Headerfile:
  - typedef entfernen.
  - Die Funktionsprototypen in den struct vor die Daten kopieren.
  - Zwischen Funktionen und Daten ggf. private ergänzen
  - Den ersten Parameter bei allen Funktionen entfernen.
  - Die create-funktion entfernen.
- Das Ergebnis könnte wie folgt aussehen:

Voila! Unsere erste Klassendeklaration!

```
#ifndef _H_BTIME_
#define _H_BTIME_

struct clbtime
{
  void setH(int h);
  void setM(int m);
  void getCurrentTime();
  void show();
  void increment();
private:
  int h;
  int m;
};

#endif
```

Öffentliche Funktionen

Private Daten

Nun zum Implementationsfile

- Streichen des jeweils ersten Parameters
- Vor den Funktionsnamen den Klassennamen, gefolgt von :: setzen (hier clbtime::)
- Den ersten Parameter, der im Beispiel immer ein Pointer vom Typ btime* war, im Anweisungsteil durch this-> ersetzen.
- Noch ein paar Kleinigkeiten:
  - malloc braucht jetzt einen cast
  - Includes auf cpp-includes ändern
  - printf auf cout<< ändern

```
#include <iostream>
#include <iomanip>
#include <cstdio>
#include <ctime>
#include <cstdlib>
using namespace std;

#include "clbtime.h"

void clbtime::setH(int h)
     {this->h=h;}
void clbtime::setM(int m)
      {this->m=m;}

void clbtime::getCurrentTime()
{

  time_t lc = time(NULL);
  struct tm*plc = localtime(&lc);
  int h,m;
  this->h=plc->tm_hour;
  this->m=plc->tm_min;
}
```

```
void clbtime::show()
{
  cout<<setw(2)<<setfill('0')
  <<this->h
  <<':'<<setw(2)<<setfill('0')
  <<this->m;
}

void clbtime::increment()
{
  this->m++;
  if (this->m==60)
    {this->m = 0; this->h++;}
  if (this->h==24)
    {this->h = this->m = 0;}
}
```

Anmerkungen zu this

- Hinter this verbirgt sich in den Memberfunktionen ein Poiner auf das Objekt, zu dem die Memberfunktion aufgerufen worden ist.
- Man kann this-> beim Zugriff auf Member auch weglassen. In manchen Situationen braucht man this aber, z.B. bei Namensgleichheit von Funktionsparameter(n) und Member(n).

```
this->p=p;
```

Änderungen im Hauptprogramm

- Änderungen der includes
- Statt des Pointers definieren wir uns eine Variable `clbtime bt;`
- Bei allen Funktionsaufrufen, die unsere Klasse betreffen, streichen wir den ersten Parameter.
- Vor alle Funktionsaufrufe , die unsere Klasse betreffen, setzen wir `bt.` , damit verdeutlichen wir, dass wir die Funktion zu dem Objekt bt aufrufen, und bewirken dass der interne Pointer `this` genau auf dieses Objekt zeigt.

Verwendung unserer ersten Klasse

```
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <iomanip>
#include <ctime>
using namespace std;
#include "clbtime.h"
```

```
int main(int argc, char*argv[])
{
  clbtime bt;
  if (argc==3)
    {bt.setH(atoi(argv[1]));
     bt.setM(atoi(argv[2]));}
  else bt.getCurrentTime();
  cout<<"time is now  ";
  bt.show();
  cout<<endl;
  bt.increment();
  cout<<"next time is ";
  bt.show();
  cout<<endl;
  return 0;
}
```

Klassen und Objekte

- Eine Klasse definiert einen benutzerdefinierten Datentyp.
- In der Regel stellt eine Klasse einen Verbund aus Daten (Membervariablen) und Funktionalität (Memberfunktionen/Methoden) dar.
- Instanzen einer Klasse bezeichnet man als Objekte.
- Objekte sind gekennzeichnet durch:
  - State (Status): Gesamtheit der Werte der Membervariablen
  - Behavior (Verhalten): Bestimmt durch die Gesamtheit der Memberfunktionen

Klassen und Objekte

- Es gibt also public Member und private Member.
- In C++ gibt es in Klassen public- und private Bereiche. Sie werden durch `public:` bzw. `private:` eingeleitet und gelten bis zur nächsten Festlegeung oder dem Ende der Klassendeklaration. Public und private regeln die Sichtbarkeit von Membern.
- Wird eine Klasse mit dem Schlüsselwort `struct` , wie in der Beispielklasse clbtime eingeleitet, so sind die Member, solange nichts anderes festgelegt ist, public. Beginnt eine Klasse mit dem Schlüsslwort `class`, so sind die Member vor einer definierten Festlegung private.
- struct oder class nennt man Klasskey.
- Die Klasse clbtime könnte auch, wie folgt lauten:

Klassen und Objekte

```
#ifndef _H_BTIME_
#define _H_BTIME_

class clbtime
{
public:
  void setH(int h);
  void setM(int m);
  void getCurrentTime();
  void show();
  void increment();
private:
  int h;
  int m;
};

#endif
```

Initialisierung

- Zur Initialisierung von Objekten gibt es spezielle Funktionen – Constructoren.
- Ein Constructor trägt den Namen der Klasse als Funktionsname und hat keinen Returntyp.
- Ein Constructor der Klasse clbtime könnte z.B. folgenden Prototyp haben:

```
clbtime();                    //in der Classdekaration
clbtime::clbtime(){...} // im Implementationsfile
```

- Constructoren können überladen werden, eine Klasse kann also mehrere Contructoren haben.
- Die Daten eines Objektes sollen immer valide sein. Dafür hat der u.a. Constructor zu sorgen.

Constructoren

```
class clbtime
{
public:
   clbtime();
   clbtime(int h, int m);
   static clbtime* create();
   void setH(int h);
   void setM(int m);
   void getCurrentTime();
   void show();
   void increment();
private:
   int h;
   int m;
};
```

include und defines
Wie gehabt, deshalb hier weggelassen

Constructoren

```
#include "clbtime.h"

// Defaultconstructor
clbtime::clbtime()
{
  h=m=0;
}

// Ueberladener Constructor
clbtime::clbtime(int h, int m)
{
  if (h>0 && h<24)  this->h=h;
  else              this->h=0;
  if (m>0 && m<59)  this->m=m;
  else              this->m=0;
}

. . .
```

Ausschnitt aus dem Implementstionsfile

Constructoren

```
int main(int argc, char*argv[])
{
  clbtime bt1;
  clbtime bt2(12,00);
  cout<<"time bt1:  ";
  bt1.show();
  cout<<endl;
  cout<<"time bt2:  ";
  bt2.show();
  cout<<endl;
}
```

Includes, wie gehabt

Aufruf des Defaultconstructors

Aufruf des Überladenen Constructors

```
./a.out
time bt1:  00:00
time bt2:  12:00
```

Constructoren

- Ein Constructor wird automatisch bei der Erzeugung eines Objektes aufgerufen.
- Gibt es mehrere, überladene Constructoren, so wählt der Compiler auf Grundlage der angegebenen Parameter den passenden Constructor aus.
- Constructoren können auch Defaultargumente haben, es dürfen nur keine Mehrdeutigkeiten entstehen.
- Hat eine Klasse keinen Constructor, so generiert der Compiler einen Defaultconstructor ohne jedwede Funktionalität. Die Member sind in diesem Falle mit undefinierten Werten belegt!

Constructoren

- Ein Constructor tätigt normalerweise keine Ein- oder Ausgaben.
- Constructoren müssen public sein, sonst sind sie sinnlos.
- die Parameter einer Instanzierung einer Klasse (Objekterzeugung) sind die Parameter des entsprechenden Konstruktors.

Memberfunktionen/Methoden

- Memberfunktionen könnn auch überladen werden und ebenfalls Defaultparameter haben.
- Memberfunktionen eine Klasse können sich gegenseitig durch this->func() aufrufen, wobei func hier für die aufzurufende Funktion steht und natürlich Parameter haben kann. Die Angabe von this-> kann aber in der Regel auch entfallen.
- Sehr kleine Memberfunktionen (getter/setter -funktionen) können in der Klassendeklaration ausprogrammiert werden. Sie werden dann als Inlinefunktionen übersetzt.

Memberfunktionen/Methoden

```
#ifndef _H_BTIME_
#define _H_BTIME_

class clbtime
{
public:
   clbtime();
   clbtime(int h, int m);
   static clbtime* create();
   void setH(int h){if (h>0 && h<24)this->h=h;}
   void setM(int m){if (m>0 && m<59)this->m=m;}
   int getm(){return m;}
   int geth(){return h;}
   void getCurrentTime();
   void show();
   void increment();
private:
   int h;
   int m;
};

#endif
```

Diese Funktionen tauchen dann im Implementationsfile nicht mehr auf

Memberfunktionen/Methoden

- Memberfunktionen sollen so implementiert sein, dass die Daten eines Objektes immer valide Werte beinhalten.
- Memberfunktionen kann man funktional in Verwaltungs-funktionen, Implementierungsfunktionen, Hilfsfunktionen und Zugriffsfunktionen einteilen.
- Gültigkeitsbereiche in Memberfunktionen
  - lokaler Block, in dem der Bezeichner verwendet wird
  - umfassende Blöcke innerhalb der Funktion, in der der Bezeich-ner verwendet wird
  - Klasse, in der die Funktion als Memberfunktion deklariert wor-den ist
  - Quelldatei (Filescope)
  - wird ein Bezeichner in einem eingeschlossenen Gültigkeitsbe-reich erneut vereibart, so verdeckt diese Vereinbarung die ur-sprüngliche Vereinbarung.

Der Destructor

- Eine Klasse kann einen Destructor definieren.
- Der dient dem Abbau eines Objektes, insbesondere dann, wenn es, meist über Pointer, andere Ressourcen bindet oder verwaltet.
- Der Name des Destrutors wird aus dem Klassennamen und einer vorgestellten Tilde (~) gebildet, er hat keinen Returntyp, keine Parameter und kann demzufolge auch nicht überladen werden.
- Er wird automatisch aufgerufen, wenn die Lebenszeit eines Objektes, auf welche Weise auch immer, endet.
- Nicht alle Klassen brauchen einen Destructor.

Referenzen als Returnwert

- Getter sollen keine Referenzen auf Memberdaten zurückgeben.
- Eine get-Funktion, die eine Referenz zurückgibt, eröffnet die Möglicheit, Daten im Objekt unkontrolliert zu manipulieren.
- Eine get-Funktion der nachfolgenden Form:

```
int& getH(){return h;}
```

ermöglicht nachfolgende Anweisung:

```
bt1.getH()=88;
```

```
./a.out
time bt1:   88:00
time bt2:   12:00
```

- Lösung: Referenzen auf Memberdaten als Returnwert vermeiden oder const-Referenzen verwenden.

Const Objekte

- C++ getstattet es, auch von Klassen Objekte zu definieren

  const clbtime start(12,00);

- Für solche Const-Objekte gelten Einschränkungen.
- Da Memberdaten durch Memberfunktionen verändert werden können, dies für const-Objekte aber nicht zulässig ist, verbietet der Compiler den Aufruf aller Memberfunktionen.

Const Objekte

- Einzelne Memberfunktionen, die die Memberdaten nicht verändern, können durch ein dem Funktionskopf nachgestellen `const`, davon ausgenommen werden.
- Der Compiler lässt diese Funktionen dann auch für Aufrufe zu const-Objekten zu.
- Der Compiler überprüft aber auch, dass keine Member in diesen Funktionen als LValue verwendet werden.
- Funktionen mit/ohne const können bei sonst gleicher Parameterliste überladen werden.