# Prüfung C/C++ (10.01.2005)

| | |
|---|---|
| Vorname | *Klaus* |
| Nachname | *Wachtler* |
| Matrikelnummer | *12345678* |

## Allgemeine Hinweise

Die Fragen sind durch freien (*aber bitte lesbaren*) Text im entsprechenden Feld zu beantworten. Der vorgesehene Platz ist **NICHT** bindend: Einerseits kann man manche Fragen auch sinnvoll beantworten, ohne den ganzen Platz zu benötigen; andererseits kann man auch auf einem zusätzlichen Blatt fortfahren, wenn man einen Roman schreiben will. Dann aber die entsprechende Antwort mit der Nummer der Frage kennzeichnen!

Bitte nur mit dokumentenechten Stiften schreiben (**kein Bleistift!**), sowie nicht mit roter (oder ähnlicher) Farbe.

**Nicht deutlich lesbare Antworten werden nicht gewertet!**

*Jedes zusätzliche Blatt* bitte mit Name und Matrikelnummer versehen, und spätestens bei der Abgabe mit den ausgeteilten Blättern zusammenheften.

## Im Notfall fragen

Wenn eine Frage unklar oder falsch formuliert ist, dann lieber sofort fragen, anstatt Zeit zu verschwenden.

Wer eine Wissenslücke hat, die man schnell beheben könnte, und wegen dieser Lücke den Rest einer Aufgabe nicht lösen kann, kann sich einen Hinweis geben lassen, der mit dem Anschreiben angemessener Strafpunkte einhergeht. Damit kann er dann hoffentlich den Rest der Aufgabe lösen und Folgefehler vermeiden.

**Und jetzt bitte beruhigen, tief Luft holen, und viel Glück!**

Anmerkungen zu den in rot angegebenen Bewertungen:

- Die Punktzahlen sind Richtwerte, anhand derer ich die Klausuren korrigiert habe, um mehrfach vorkommende positive oder negative Bewertungen einheitlich zu gestalten. Wer richtige Lösungen angibt, die von der Musterlösung abweichen, bekommt natürlich analog Punkte.
- Im Einzelfall weiche ich davon etwas ab: wenn mir irgendetwas an einer Antwort sagt, daß jemand den Stoff überhaupt nicht verstanden hat, dann gibt es auch für Zufallstreffer keine Punkte. Ist dagegen etwas faktisch falsch, aber es geht aus dem Zusammenhang hervor daß jemand es verstanden hat, dann gibt es trotzdem noch den einen oder anderen Punkt.
  Vollkommen willkürlich also.
- Bei Folgefehlern bemühe ich mich, sparsam abzustrafen.
- Wenn etwas nicht zu entziffern ist oder nur zusammenhanglos irgendwelche Stichworte herumstehen, ignoriere ich den entsprechenden Teil (ich weiß, daß mein Tafelbild auch gelegentlich schwer verständlich ist, aber das wird an anderer Stelle bewertet).
  Bei Kandidaten mit anderen Muttersprachen bin ich natürlich wesentlich geduldiger; aber letztlich kann ich auch da nur bewerten was mit gutem Willen nachzuvollziehen ist.

Schon mal vorab für eventuelle Einsichtnahme: teilweise habe ich (wenn wenig vorhanden ist) von 0 ausgehend nach Bewertbarem gesucht und Punkte addiert; ist die Antwort dagegen im Wesentlichen brauchbar, gehe ich von der Maximalpunktzahl aus und mache ggf. Abstriche. Innerhalb einer Aufgabe wechselt das auch noch gelegentlich.
Negative Punkte in den Originalklausuren sind also Abzüge von der Maximalpunktzahl, während sich positive auf die 0 beziehen.

## Prüfungsfragen

1. In ANSI-C soll eine einfach verkettete Liste irgendwelcher Nutzdaten aufgebaut werden (nullterminierte Zeichenketten).

   Jeweils ein Listenelement soll in einer `struct` der Art:

   ```
   typedef struct Listenelement
   {
       char                   nutzdaten[100];
       struct Listenelement  * next;
   } Listenelement_t;
   ```

   abgelegt werden.

   Der Aufrufer soll einen Zeiger auf das erste Listenelement als automatische Variable haben:

   ```
   int main()
   {
      struct Listenelement  * listenanfang = NULL;
      // ...
   }
   ```

   Aufgabe: Es soll eine Funktion `einfanf()` angegeben werden, welche

   - in einer geeigneten Art und Weise den Listenanfang übergeben bekommt (in der Parameterliste),
   - eine nullterminierte Zeichenkette (die von der Funktion nicht verändert wird) übergeben bekommt,
   - die übergebene Zeichenkette als neues Element an den Anfang der Liste einfügt,
   - als Rückgabewert liefert, ob das Einfügen geklappt hat

   Die Funktion soll auch bei einer leeren Liste erfolgreich einfügen!

Für die Liste muß die Adresse des Listenanfangs beim Aufrufer übergeben werden, deshalb Zeiger auf Zeiger auf `Listenelement_t`; ansonsten kann man den Listenanfang beim Aufrufer nicht ändern. Die Änderung über eine Zuweisung beim Aufrufer habe ich extra unterbunden, indem Erfolg/Mißerfolg als Rückgabewert verlangt ist (5).
`malloc` und Rückgabewert testen (1).
Listenelement richtig einhängen (2).
Nutzdaten korrekt kopieren (1)
Test der Daten (1)
`#include` (1)
`int` als Rückgabewert und korrekte Rückgabe (1)

**Summe: 12**

```
/* gekürzte Version der Musterlösung zu Aufgabe 4
 * fügt am Anfang der Liste ein Element ein, bzw fügt erstes
 * Element ein, falls Liste leer ist.
 * Rückgabewert ist 1, falls alles geklappt hat, oder 0,
 * falls kein Speicher allokiert werden konnte.
 * Von den Daten wird beim Einfügen eine Kopie gemacht
 * (Wertsemantik).
 */

#include <stdlib.h>
#include <string.h>


/* 1*/ int einfanf( Listenelement_t **lanf_p,
/* 2*/                const char      *daten
/* 3*/                )
/* 4*/ {
/* 5*/   /* neues Element allokieren: */
/* 6*/   Listenelement_t *neuesElement
/* 7*/     = (Listenelement_t*)malloc( sizeof(Listenelement_t) );
/* 8*/   if( !neuesElement )
/* 9*/   {
/*10*/     fprintf( stderr, "kein Speicher in einfanf()\n" );
/*11*/     return 0; /* gescheitert */
/*12*/   }
/*13*/
/*14*/   /* neues Element füttern: */
/*15*/   strncpy( neuesElement->daten,
/*16*/            daten,
/*17*/            sizeof(neuesElement->nutzdaten)-1
/*18*/            );
/*19*/   /* sicherheitshalber terminieren: */
/*20*/   neuesElement->daten[sizeof(neuesElement->daten)-1] = '\0';
/*21*/
/*22*/   neuesElement->next = *lanf_p; /* Alter LA wird Nachfolger */
/*23*/
/*24*/   /* Neues Element als Listenanfang speichern: */
/*25*/   *lanf_p = neuesElement;
/*26*/   return 1;   /* erfolgreich */
/*27*/ }
```

2. Wie sieht für die vorige Aufgabe der Aufruf der Funktion `einfanf()` aus (bitte mit Prüfung des Rückgabewerts!)?

```
struct Listenelement  * listenanfang = NULL;
/* ... */

if( einfanf( &listenanfang, "blabla" ) )
{
    /* Einfügen hat geklappt; mit
     * der neuen Liste arbeiten:
     */
}
else
{
    /* Einfügen gescheitert; Fehler ausgeben... */
}
```

Hier muß die Adresse des Listenanfangs übergeben werden!
Korrekter Aufruf (2)
Rückgabewert testen (1)

**Summe: 3**

---

3. Bitte beschreiben Sie nachvollziehbar, was beim Aufruf Ihrer Funktion `einfanf()` aus den vorigen beiden Aufgaben passiert...

   (a) ...für den Fall, daß die Liste beim Aufruf noch leer ist:

wichtig ist mir hier, daß aus der Beschreibung klar wird, wie der Listenanfang beim Aufrufer korrigiert wird (4)
Restlicher Vorgang (4)

*Beim Aufruf wird als erster Parameter die* **Adresse** *der Variablen* `listenanfang` *des Aufrufers übergeben, sowie die einzufügenden Daten (als Zeiger auf das erste Zeichen eines nullterminierten Strings).*

*In* `einfanf()` *ist der Parameter* `lanf_p` *die übergebene Adresse;* `*lanf_p` *ist die Variable* `listenanfang` *des Aufrufers (und anfangs* `NULL`*, weil die Liste ja noch leer ist).*

*In Zeile 7 wird mit* `malloc()` *Speicher für das neue Listenelement beschafft, und der Zeiger darauf erstmal in* `neuesElement` *geparkt (hat* `malloc() nicht geklappt, wird die Funktion beendet und 0 zurückgegeben`*).*

*In Zeile 15 werden die gewünschten Nutzdaten in das neu allokierte Element kopiert (hier mit* `strncpy()`*, was sicherer ist als* `strcpy()`*; dafür wird in Zeile 20 der String terminiert für den Fall, daß er zu lang war).*

*Als Nachfolger des neuen Elements wird in Zeile 22 über den Zeiger* `lanf_p` *die Variable* `listenanfang` *des Aufrufers eingetragen, also der Wert* `NULL`*, weil die Liste vorher leer war.*

*In 25 wird über den Zeiger* `lanf_p` *der Listenanfang des Aufrufers auf das neue Element gerichtet.*

*Wenn bis hier alles geklappt hat, wird als Erfolgsmeldung* `1` *zurückgegeben.*

**Summe: 8**

(b) …für den Fall, daß die Liste beim Aufruf bereits zwei Elemente enthält:

*Fast alles wie in der vorigen Aufgabe, aber als Nachfolger des neuen Elements wird in Zeile 22 über den Zeiger* `lanf_p` *(alias die Variable* `listenanfang` *des Aufrufers) nicht* `NULL`*, sondern ein Zeiger auf das bisherige erste Element eingetragen.*

Für den korrekten Vergleich der Unterschiede (bzw. Nichtunterschiede :-) zu 3a: (3) (Falls Antwort zu 3a nicht existiert, aber hier voll beschrieben wird, entsprechend Punkte wie sonst in 3a)

**Summe: 3**

4. Es soll ein `template` erstellt werden, welches ein Feld von Werten beliebigen Typs repäsentiert (mit einem Index ab 0).

   Im Konstruktor soll beim Anlegen die Anzahl der Elemente angegeben werden.

   Bitte ergänzen Sie den folgenden Quelltext entsprechend, sodaß die anschließend gezeigte Verwendung sinnvoll möglich ist:

Korrekte Struktur (3)
ctor ok (2)
dtor ok(2)
operator[] (2)
Feldgrenzen (1)
ggf. Abzug für unsinnige Methoden (-1)

**Summe: 10**

```
// Time-stamp: "12.01.05 12:46 tarr.cpp klaus@wachtler.de"

#include <iostream>

template <typename T> class Arr                           // auszufüllen
{                                                         // auszufüllen
public:                                                   // auszufüllen
  Arr( int anzahl )                                       // auszufüllen
    : arr( new T[anzahl] ),                               // auszufüllen
      anzahl( anzahl )                                    // auszufüllen
  {}                                                      // auszufüllen
                                                          // auszufüllen
  ~Arr()                                                  // auszufüllen
  {                                                       // auszufüllen
    delete[] arr;                                         // auszufüllen
  }                                                       // auszufüllen
                                                          // auszufüllen
  T &operator[]( int index ) const                        // auszufüllen
  {                                                       // auszufüllen
    if( index<0 || index>=anzahl )                        // auszufüllen
    {                                                     // auszufüllen
      std::cerr << "Index falsch" << std::endl;           // auszufüllen
      exit( 1 );                                          // auszufüllen
    }                                                     // auszufüllen
    return arr[index];                                    // auszufüllen
  }                                                       // auszufüllen
                                                          // auszufüllen
private:                                                  // auszufüllen
  int   anzahl;                                           // auszufüllen
  T   * arr;                                              // auszufüllen
};                                                        // auszufüllen
                                                          // auszufüllen
                                                          // auszufüllen
                                                          // auszufüllen
                                                          // auszufüllen
```

```
int main( int nargs, char **args )
{

  Arr<int>   einFeld( 10 );

  einFeld[0] = 10;
  einFeld[5] = 5;
  einFeld[9] = einFeld[0] + einFeld[9];

  Arr<double>   einanderesFeld( 10 );

  einanderesFeld[0] = 10.5;
  einanderesFeld[5] = 5.9;
  einanderesFeld[9] = einanderesFeld[0] + einanderesFeld[9];

  std::cout << einanderesFeld[9] << std::endl;

  return 0;
} // main( int nargs, char **args )

// Ende von tarr.cpp
```

5. Was gibt das folgende Programm aus (bitte mit Begründung!)?

```
// Time-stamp: "10.01.05 22:04 ausgabe.cpp klaus@wachtler.de"

#include <iostream>
#include <ostream>

class A
{
public:

  // Konstruktor
  A()
    : ersterwert( 0 )
  {
    std::cerr << "ctor A()" << std::endl;
  }

  A( int ersterwert )
    : ersterwert( ersterwert )
  {
    std::cerr << "ctor A( " << ersterwert << " )"
              << std::endl;
  }

  // copy-Konstruktor
  A( const A &rechteSeite )
    : ersterwert( rechteSeite.ersterwert )
  {
    std::cerr << "copy-ctor A( const A &rechteSeite )"
              << " rechteSeite.ersterwert = "
              << rechteSeite.ersterwert
              << std::endl;
  }

  // Destruktor
  virtual ~A()
  {
    std::cerr << "dtor ~A(), " << ersterwert << std::endl;
  }

  // Zuweisung A = A
  // Rückgabewert ist eine REFERENZ auf die linke Seite
  // (nicht const), und damit wieder als lvalue verwendbar.
  A &operator=( const A &rechteSeite )
  {
    std::cerr << "Zuweisung A=A, ls=" << ersterwert
```

```
              << ", rs=" << rechteSeite.ersterwert
              << std::endl;

    ersterwert = rechteSeite.ersterwert;
    return *this;
  }

  // Ausgabe auf einen ostream
  virtual void streamout( std::ostream & s ) const
  {
    s << "A=" << ersterwert;
  }

  // Kopie von sich selbst allokieren
  virtual A *clone() const
  {
    std::cout << "A::clone()" << std::endl;
    return new A(*this);
  }

protected:

  // Membervariable (in den abgeleiteten Klassen kommen noch welche
  // dazu):
  int ersterwert;

}; // class A...


class B: public A
{
public:

  // Konstruktor
  B()
    : A(),
      zweiterwert( 0 )
  {
    std::cerr << "ctor B()" << std::endl;
  }

  B( int ersterwert, int zweiterwert )
    : A( ersterwert ),
      zweiterwert( zweiterwert )
  {
    std::cerr << "ctor B( "
              << ersterwert
```

```
              << ", "
              << zweiterwert
              << " )"
              << std::endl;
}

// copy-Konstruktor
B( const B &rechteSeite )
  : A( rechteSeite.ersterwert ),
    zweiterwert( rechteSeite.zweiterwert )
{
  std::cerr << "copy-ctor B( const B &rechteSeite )"
            << "\n  rechteSeite.ersterwert = "
            << rechteSeite.ersterwert
            << ",\n  rechteSeite.zweiterwert = "
            << rechteSeite.zweiterwert
            << std::endl;
}

// Destruktor
virtual ~B()
{
  std::cerr << "dtor ~B()"
            << ", ersterwert=" << ersterwert
            << ", zweiterwert=" << zweiterwert
            << std::endl;
}

// Zuweisung B = B
// Rückgabewert ist eine REFERENZ auf die linke Seite
// (nicht const), und damit wieder als lvalue verwendbar.
B & operator=( const B &rechteSeite )
{
  std::cerr << "Zuweisung B=B"
            << ",\n  ls.ersterwert=" << ersterwert
            << ",\n  ls.zweiterwert=" << zweiterwert
            << ",\n  rs.ersterwert=" << rechteSeite.ersterwert
            << ",\n  rs.zweiterwert=" << rechteSeite.zweiterwert
            << std::endl;

  this->A::operator=( rechteSeite );
  zweiterwert = rechteSeite.zweiterwert;

  return *this;
}

// Ausgabe auf einen ostream
```

```
  virtual void streamout( std::ostream & s ) const
  {
    s << "B=";
    A::streamout( s );
    s << ", "  << zweiterwert;
  }

  // Kopie von sich selbst allokieren
  virtual B *clone() const
  {
    std::cout << "B::clone()" << std::endl;
    return new B(*this);
  }

protected:

  // Membervariable
  int zweiterwert;

}; // class B...

inline std::ostream &operator<<( std::ostream &s, const A &obj )
{
  obj.streamout( s );
  return s;
}

int main( int nargs, char **args )
{
  std::cout << "es geht los..." << std::endl;
  B einb( 12, 20 );
  std::cout << "new:" << std::endl;
  B *p_b = new B( 1, 2 );
  std::cout << "Ausgabe:" << std::endl;
  std::cout << "*p_b = " << *p_b << std::endl;
  B nocheinb = einb;
  std::cout << "nocheinb = " << nocheinb << std::endl;
  nocheinb = *p_b;
  std::cout << "nocheinb = " << nocheinb << std::endl;
  std::cout << "clone:" << std::endl;
  A *p_keineahnung = &einb;
  A *p_keineahnung2 = p_keineahnung->clone();
  std::cout << "*p_keineahnung2 = " << *p_keineahnung2 << std::endl;
  std::cout << "fertig!" << std::endl;
} // main( int nargs, char **args )
```

Hier muß man natürlich wissen, daß bei Zeigern und Referenzen Methoden, die als virtual deklariert sind, aus dem tatsächlichen Objekt aufgerufen werden, auch wenn der Zeiger oder die Referenz als eine Basisklasse davon deklariert ist. Dementsprechend will ich in der Begründung erkennen können, für welches Objekt welche Methode aufgerufen wird. Konstruktoren von Basisklassen werden vor dem Rumpf der abgeleiteten Klasse aufgerufen; beim dtor umgekehrt. Teilpunkte jeweils ggf. (Ausgabe + Beschreibung)

*(ich habe i.F. die Programmausgabe in blau gehalten, die Erklärungen dazu in grün.)*

`es geht los...` *(0)*

`ctor A( 12 )` *weil einb.B::B(int,int) gleich zu Anfang einb.A::A(int) mit seinem ersten Parameter aufruft*

`ctor B( 12, 20 )` *das einb.B::B(int,int), welches eben schon einb.A::A(int) aufrief (1+1)*

`new:` *(0)*

`ctor A( 1 )` *wieder indirekt über p_b->B::B(int,int)*

`ctor B( 1, 2 )` *zusammen (1)*

`Ausgabe:` *(0)*

`*p_b = B=A=1, 2` *operator<< wird mit einer Referenz auf *p_b aufgerufen, also einem B. Weil streamout() virtuell ist, wird p_b->B::streamout aufgerufen (Ausgabe von B=), darin kommt dann ein Aufruf von p_b->A::streamout (Ausgabe von A=1); der Rest von p_b->B::streamout schreibt dann das Komma und die 2 (2+2)*

`ctor A( 12 )` *für einb indirekt über den copy-ctor aus B*

```
copy-ctor B( const B &rechteSeite )
  rechteSeite.ersterwert = 12,
  rechteSeite.zweiterwert = 20
```
*Initialisierung von nocheinb mit einb; Achtung: hier ist keine Zuweisung, trotz des = (2+2)*

`nocheinb = B=A=12, 20` *wie oben: operator<< für einb ruft einb.B::streamout auf, das wiederum einb.A::streamout aufruft (1)*

```
Zuweisung B=B,
  ls.ersterwert=12,
  ls.zweiterwert=20,
  rs.ersterwert=1,
  rs.zweiterwert=2
```
*es wird nocheinb.B::operator= aufgerufen mit *p_b als Parameter*

`Zuweisung A=A, ls=12, rs=1` *darin wiederum wird nocheinb.A::operator= aufgerufen mit *p_b als Parameter bis hier (2+2)*

`nocheinb = B=A=1, 2` *Ausgabe von nocheinb wie oben (1)*

`clone:` *jetzt wird es spannend... (0)*

*An p_keineahnung wird die Adresse von einb zugewiesen (oder initialisiert, hier egal). Davon sind keine Objekte betroffen, weil es nur eine Zeigeroperation ist. Keine Methodenaufrufe, keine Ausgabe. Aber jetzt enthält der A-Zeiger p_keineahnung enthält jetzt einen Zeiger auf ein B, nämlich auf einb. (0+2)*

`B::clone()` *clone ist eine virtuelle Methode; weil p_keineahnung auf ein B zeigt (das einb) und clone virtuell ist, wird erstmal einb.B::clone aufgerufen. Dieses macht ein new B(*this), wodurch ein neues B allokiert und der copy-ctor aus B aufgerufwen wird, ... (1+3)*

`ctor A( 12 )` *... der zuerst A::A( int ) mit seinem ersten Parameter aufruft, ...*
`copy-ctor B( const B &rechteSeite )` *... dann kommt sein eigener Rumpf.*
  `rechteSeite.ersterwert = 12,`
  `rechteSeite.zweiterwert = 20` *p_keineahnung2 bekommt die Adresse des neu allokierten B zugewiesen.* *bis hier (3+3)*
`*p_keineahnung2 = B=A=12, 20` *Weil p_keineahnung2 auf ein B zeigt, dieses als Referenz an operator<< übergeben wird, dort damit die* **virtuelle** *Methode streamout auf gerufen wird, wird ein B ausgegeben; siehe oben (p_keineahnung2->B::streamout ruft p_keineahnung2->A::streamout auf).* *(1)*
`fertig!` *natürlich nicht; es müssen noch lokale Variablen entsorgt werden!* *(0)*
`dtor ~B(), ersterwert=1, zweiterwert=2` *für nocheinb wird am Blockende der dtor nocheinb.B::~B aufgerufen, ...*
`dtor ~A(), 1` *..., der automatisch den dtor aus A aufruft (also nocheinb.A::~A)*
`dtor ~B(), ersterwert=12, zweiterwert=20` *für einb wird am Blockende der dtor einb.B::~B aufgerufen, ...*
`dtor ~A(), 12` *..., der automatisch den dtor aus A aufruft (also einb.A::~A)* *bis hier (3+3)*

ggf. Abzüge für zusätzliche Ausgaben
**Summe: 36**

6. Was gibt das folgende Programm aus (bitte mit Begründung!)?

```
#include <stdio.h>

char *c[]  =
  {
    "he dast ga",
    "lllt so doof",
    "C i",
    "dar nich"
  };

char **cp[] =
  {
    c+3,
    c+2,
    c+1,
    c
  };

char ***cpp  =   cp;

int main( int Anzarg, char **Arg )
{
  printf( "%s", **++cpp );
  printf( "%s", *--*++cpp+5 );
  printf( "%s", cpp[-2][0]+2 );
  printf( "%s\n", *(cpp[1]+1)+3 );
  return 0;
}
```

*Ausgabe:*
*C ist gar nicht so doof (4)*
*Die Begründung spare ich mir hier, weil sie bereits –mit einem minimal geänderten Text– im C-Skript zu finden ist; siehe* http://www.wachtler.de/ck/18_2_Losungen.html *(18 Übungen, 18.2 Lösungen, 18.2.15 Mit Zeigern jonglieren, aber richtig I) (8), aber natürlich nur, wenn nicht einfach auf das C-Skript verwiesen wird!*

**Summe: 12**