Prüfung C/C++ (25.02.2005)

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!

Bitte erst jede Aufgabe lesen, und nach dem Beantworten nochmals prüfen, ob die Antwort auch die Aufgabe erfüllt!

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. Es ist eine Funktion `strmcpy` **zusammen mit einem kleinen Testprogramm (!)** in ANSI-C zu schreiben mit folgenden Eigenschaften:

    - Als Parameter wird ein nullterminierter String übergeben, der in der Funktion nicht verändert wird.
    - die Funktion allokiert ausreichend freien Speicher, kopiert den übergebenen String in den allokierten Speicher und liefert die Kopie zurück.
    - Innerhalb dieser Funktion sollen keine Bibliotheksfunktionen zum Feststellen der Stringlänge und zum Kopieren verwendet werden (zum Allokieren natürlich schon).

    Bitte außerdem auf folgende Punkte achten:

    - Der Quelltext soll sinnvoll gestaltet sein (angemessene Kommentare, sinnvolle Namen, Übersichtlichkeit)
    - Nötige Headerdateien nicht vergessen!
    - Fehlermöglichkeiten sollen sauber kontrolliert und abgefangen werden.
    - Resourcen sollen sauber freigegeben werden
    - Das Testprogramm nicht vergessen!

Kommentar Programm (1)

```c
/* Time-stamp: 24.02.05 20:32 strmcpy.c
 *
 * Testprogramm für strmcpy()
 * Musterlösung für die erste Aufgabe der  C-/C++-Klausur 25.02.2005
 * FH Hannover
 *
 * Erstellt 24.02.2005 20:12 Klaus Wachtler
 *
 */
```

include (2)

```c
#include <stdlib.h>
#include <stdio.h>
#include <stddef.h>
```

Kommentar Funktion (1)

Kommentar Fehlerbehandlung (1)

Kommentar Freigabe (1)

Korrekte Signatur (1)

```c
/* strmcpy() allokiert ausreichend Speicher für eine Kopie des
 * übergebenen Parameters, kopiert den übergebenen String dorthin und
 * liefert einen Zeiger darauf zurück.
 * Im Fehlerfall (kein Speicher frei, oder übergebener String ist
 * NULL) wird NULL zurückgegeben.
 * Die Freigabe des allokierten Speichers muß beim Aufrufer erfolgen.
 */
char *strmcpy( const char* src_string )
{
  char        *kopie = NULL; /* Zeiger auf Kopie des Strings          */
  size_t       laenge = 0;   /* Später die Länge inkl. abschl. 0      */
  size_t       i;            /* Schleifenzähler                       */
```

Test der Parameter (1)

```c
  /* Parameter testen: */
  if( !src_string )
  {
    return NULL;
  }
```

Länge korrekt bestimmen (2)

```c
  /* Länge von src_string feststellen: */
  i = 0;
  do /* nichtabweisende Schleife, d.h. abschl. 0 wird mitgezählt */
  {
    laenge++;
  }
  while( src_string[i++] );
```

Allokieren (2)

```c
  /* Speicher für Kopie allokieren */
  if( !(kopie = (char *)malloc( laenge )) )
  {
```

```
    /* kein Speicher frei! */
    return NULL;
  }

  /* String kopieren (inkl. abschl. 0): */
  for( i=0; i<laenge; i++ )
  {
    kopie[i] = src_string[i];
  }

  return kopie;
}

int main( int nargs, char **args )
{
  /* Ein String wird zweimal kopiert.
   * Beim ersten Mal wird eine Konstante kopiert, beim zweiten Mal ist
   * es bereits ein frei allokierter Speicher.
   */
  char *testkopie1;
  char *testkopie2;

  if( (testkopie1=strmcpy( "blabla weö ,depc,e pep" )) )
  {
    if( (testkopie2=strmcpy( testkopie1 )) )
    {
      printf( "Kopie der Kopie =<%s>\n", testkopie2 );
      free( testkopie2 );
    }
    else
    {
      printf( "Erste Kopie mißlungen!\n" );
    }
    free( testkopie1 );
  }
  else
  {
    printf( "Erste Kopie mißlungen!\n" );
  }

  return 0;
}
```

Rückgabe testen (1)

Kopieren+Terminieren (2)

Testprogramm inkl. Rückgabe testen (3)

Summe: 18

2. Es sollen `template`-Funktionen geschrieben werden mit den Namen `min` und `max`, die für wahlweise für 2 oder 3 Parameter den kleinsten (`min`) bzw. den größten (`max`) Wert der Parameter liefern; bei jedem Aufruf haben alle Parameter denselben Typ.

   Also: `min` mit zwei Parametern gleichen Typs liefert den kleineren der beiden Parameter, `min` mit drei Parametern gleichen Typs liefert den kleinsten der drei Parameter.

   Analog liefert `max` den größten von zwei oder drei Parametern.

   Ihre Funktionen sollen vorerst für eingebaute Datentypen (`int`, `double`, `char`...) funktionieren.

   Die Funktionen sollen in eine vollständige Headerdatei minimax.h eingebettet werden, die in einem größeren Programm (ggf. aus mehreren Quelltexten bestehend) mehrfach mit `#include` werden kann.

   Bitte auch einen kurzen Beispielaufruf mit angeben!

   (Ein vollständiges Testprogramm muß nicht angegeben werden; das folgt ohnehin in der kommenden Aufgabe.)

vernünftige Kommentare (2)

```
// Time-stamp: "24.02.05 22:41"
//
// Stellt min()- und max()-Funktionen für 2 und 3 Parameter zur
// Verfügung.
//
// 24.02.2005 Klaus Wachtler
//              (Musterlösung für die C-/C++-Klausur 25.02.2005)
```

mehrfaches Include erlauben (2)

```
#ifndef _MINIMAX_H_
#define _MINIMAX_H_
```

const-Referenz effektiver (2)

inline effektiver und erlaubt mehrfache Definitionen (2)

```
template<typename T> inline T min( const T &a, const T &b )
{
  return a<b ? a : b;
}

template<typename T> inline T min( const T &a, const T &b, const T &c )
{
  return min( a, min( b, c ) );
}

template<typename T> inline T max( const T &a, const T &b )
{
  return b<a ? a : b;
}

template<typename T> inline T max( const T &a, const T &b, const T &c )
```

```
{
  return max( a, max( b, c ) );
}

#endif /* ifdef _MINIMAX_H_ */
```

funktionsfähige templates (8)

Beispielaufruf:

```
#include "minimax.h"
...
  int i = 10, j = 13, k = 8;
  std::cout << "min = " << min( i, j, k ) << std::endl;
  std::cout << "min = " << min( 3.14, 2.718, 10.0 ) << std::endl;
```

Beispielaufruf (2)

Summe: 18

---

3. Das folgende Programm soll die Headerdatei minimax.h der vorigen Aufgabe verwenden. Es ist im Folgenden eine **unvollständige** Klasse `Student` vorgegeben.

   Im Hauptprogramm werden erst drei Objekte von `Student` definiert (`a`, `b`, `c`).

   Dann soll ein viertes Objekt `d` definiert werden und mit dem kleinsten der bisherigen drei Objekte initialisiert werden unter Verwendung Ihrer Funktion `min` aus der vorigen Aufgabe.

   Wie muß das folgende Programm ergänzt werden, damit dies fehlerfrei funktioniert?

   Es sind alle Änderungen anzugeben, damit das Programm vollständig und fehlerfrei ist.

Je nachdem, wie man in der vorigen Aufgabe die Objekte vergleicht (*kleiner*, *größer*, *kleiner gleich*, *größer gleich*), muß man hier die nötigen Operatoren überladen. Weil ich nur *kleiner* verwende, brauche ich auch nur den zu überladen.

```
// Time-stamp: "24.02.05 23:32 t.cpp klaus@wachtler.de"

#include <iostream>
#include "minimax.h"

class Student
{
public:

  Student( std::string vorname, std::string nachname, int matrikelnummer )
    : vorname( vorname ),
      nachname( nachname ),
      matrikelnummer( matrikelnummer )
  {}

  virtual ~Student()
  {}

  // Ausgabe auf einen ostream:
  virtual void streamout( std::ostream & s ) const
  {
    s << vorname;
    s << ",";
    s << nachname;
    s << ",";
    s << matrikelnummer;
  }

  bool operator<( const Student &rS ) const      // auszufüllen
  {                                              // auszufüllen
    return matrikelnummer < rS.matrikelnummer;   // auszufüllen
  }                                              // auszufüllen
```

Rückgabewerte ist `bool`, erster Operand (linke Seite) ist $*$this, rechte Seite ist der Parameter

```
protected:

  std::string  vorname;
  std::string  nachname;
  int          matrikelnummer;







};

// Ausgabe eines Student auf einen ostream:
inline std::ostream &operator<<( std::ostream &s, const Student &obj )
{
  obj.streamout( s );
  return s;
}






int main( int nargs, char **args )
{
  Student  a( "Klaus", "Wachtler", 9999 );
  Student  b( "Fritz", "Meier", 123 );
  Student  c( "", "", 789456 );

  // Hier soll der Student mit der kleinsten Matrikelnummer zum
  // Initialisieren von d verwendet werden, also Fritz Meier:
  Student  d( min( a, b, c ) );

  std::cout << "der Kleinste ist " << d << std::endl;

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

// Ende von t.cpp
```

Summe: 22

4. Was gibt das Programm `ausgabe.cpp` (siehe fünfseitige Beilage) aus?

   Bitte die einzelnen Ausgaben begründen (für welches Objekt wird welche Methode aus welcher Klasse mit welchen Parametern aufgerufen, ggf. warum diese und keine andere...)

   In Zeile 293 wird eine lokale (automatische) Variable `einb` *erzeugt mit zwei* `int`*-Parametern (30 und 40); deshalb wird dafür der ctor aus* `B` *mit zwei* `int` *aus Zeile 147 aufgerufen, also* `B::B(int,int)`*. Dieser ruft noch vor seinem eigenen Rumpf* `A::A(int)` *(Zeile 22) auf mit seinem ersten Parameter (30). Dieser gibt aus:*

   `ctor A( 30 )`

   Jetzt kommt der Rumpf von `B::B(int,int)`*:*

(2+2) `ctor B( 30, 40 )`

(0+0) `unterprogramm1:`

   Hier wird `unterprogramm1()` *aufgerufen. Dieses ist vereinbart mit einem Parameter vom Typ* `A` *(nicht als Referenz, also call by value). Es wird ein neues Objekt vom Typ* `A` *erzeugt, und mit dem vorhandenen* `einb` *(vom Typ* `B`*) initialisiert (temporäres Objekt für den Aufruf der Funktion* `unterprogramm1()`*). Für dieses temporäre Objekt wird der copy-ctor aus A aufgerufen (*`A::A( const A & )` *aus Zeile 30 mit einer Referenz auf* `einb` *als Parameter:)*

   `copy-ctor A( const A &rechteSeite ) rechteSeite.ersterwert = 30`

   Für das temporäre Objekt wird `A::tuwas()` *aufgerufen:*

(2+3) `A::tuwas: A=30`

   ebenso `A::tuwasanderes()`*:*

(1+1) `A::tuwasanderes: A=30`

   Das temporäre Objekt wird mit `operator<<` *(Zeile 94) und damit der Methode* `A::streamout()` *aus Zeile 59 ausgegeben:*

(1+1) `unterprogramm1: eina = A=30`

   Mit dem Ende von `unterprogramm1()` *wird das temporäre Objekt zerstört, also der dtor aus* `A`*:*

(1+1) `dtor ~A(), 30`

   Jetzt sind wir wieder im Hauptprogramm in Zeile 298. Mit `new` *wird ein Heapobjekt vom Typ* `B` *erzeugt; dieses wird mit dem vorhandenen* `einb` *initialisiert. Also der copy-ctor* `B::B(const B&)`*, der erst wieder den* `A::A(int)` *aufruft:*

   `ctor A( 30 )`

   Jetzt der Rumpf von `B::B(const B&)` *für das Heapobjekt*

```
copy-ctor B( const B &rechteSeite )
  rechteSeite.ersterwert = 30,
  rechteSeite.zweiterwert = 40
```

   Die Adresse des mit `new` *erzeugten* `B`*-Objekts wird an* `einanderes_p` *zugewiesen; dabei wird nur die Adresse kopiert. Daß diese Variable vom Typ* `A*` *ist, aber auf ein* `B` *zeigt, stört nicht weiter (ein Zeiger auf* `A` *darf auch auch auf etwas von* `A` *abgeleitetes zeigen).*

(3+3)

Weiter mit Zeile 300: (0+0)

`unterprogramm2:`

In Zeile 301 wird `unterprogramm2()` *aufgerufen. Dessen Parameter ist eine Referenz auf ein* `A` *oder etwas davon abgeleitetes, in diesem Fall auf das* `B` *auf dem Heap. Es wird also* **kein** *temporäres Objekt erzeugt. Vielmehr ist der Parameter* `eina` *in* `unterprogramm2()` *eine Referenz auf das Heapobjekt, also auf* `*einanderes_p`*.*

Damit wird `tuwas()` *aufgerufen, welches als* `virtual` *deklariert ist. Folglich wird für das Heapobjekt (welches ja in* `B` *ist) die Methode* `B::tuwas()` *verwendet:*

`B::tuwas: B=A=30, 40` (4+5)

`tuwasanderes()` *ist dagegen nicht als* `virtual` *deklariert. Weil der Compiler beim Aufruf nicht weiß, daß es sich in Wirklichkeit um ein* `B` *handelt, wird* `A::tuwasanderes()` *aufgerufen für das Heapobjekt. Hierin wird der Operator* `<<` *aufgerufen, der wiederum* `streamout()` *aufruft. Letzteres ist* `virtual` *deklariert, also wird für Heapobjekt (immer noch vom Typ* `B`*) die Methode* `B::streamout()` *verwendet und das gesamte Objekt ausgegeben:* (4+5)

`A::tuwasanderes: B=A=30, 40`

In 282 wird ebenso `B::streamout()` *aufgerufen (weil Referenz auf ein* `B` *und weil* `streamout()` *als* `virtual` *deklariert ist):* (1+2)

`unterprogramm2: eina = B=A=30, 40`

Jetzt wird mit `throw` *eine Konstante geworfen vom Typ* `const char*` *(Zeiger auf das erste Zeichen in* `"oioioi!"`*).*

Beim Werfen einer Ausnahme werden alle lokalen Blöcke verlassen (bis ein passendes `catch` *erreicht wird) und die darin angelegten automatischen Variablen aufgelöst. Das ist keine in* `unterprogramm2()`*, aber* `einb` *in* `main()`*. Das Heapobjekt wird nicht aufgeräumt, weil es keine automatische Variable ist. Der Zeiger darauf (*`einanderes_p`*) wird aber zerstört, und es gibt keinen Zeiger mehr auf das Heapobjekt; es kann nie mehr dessen dtor aufgerufen werden.*

Zerstören der automatischen Variable `einb`*:*

`dtor ~B(), ersterwert=30, zweiterwert=40`

Darin automatischer Aufruf des dtor des in `einb` *enthaltenen* `A`*:* (5+5)

`dtor ~A(), 30`

Durch das Werfen eines `const char*` *sind wir jetzt in 311 angekommen:* (3+3)

`anderer Fehler in main(): oioioi!`

Summe: 58, gezählt wird die Hälfte davon, also maximal 29

5. Ist das folgende Programm ein korrektes ANSI-C-Programm?
   Falls ja: Was macht es genau?
   Falls nein: Was ist daran falsch?

   In jedem Fall die Antwort nachvollziehbar begründen!
   (Nur für *ja* oder *nein* lasse ich keine zwei Seiten Platz.)

```c
/* Time-stamp: "01.03.05 09:23 falschodernicht.c klaus@wachtler.de"
 *
 */

/*  1 */ #include <stdio.h>
/*  2 */
/*  3 */ typedef  double (*p_t)( double, double );
/*  4 */
/*  5 */ double addiere( double a, double b )
/*  6 */ {
/*  7 */   return a + b;
/*  8 */ }
/*  9 */
/* 10 */ double subtrahiere( double a, double b )
/* 11 */ {
/* 12 */   return a - b;
/* 13 */ }
/* 14 */
/* 15 */ int main( int nargs, char **args )
/* 16 */ {
/* 17 */   p_t     arr[2] = { subtrahiere, addiere };
/* 18 */   p_t    *p_f = arr;
/* 19 */
/* 20 */   double ergebnis = (*++p_f)( arr[0]( 15.5, 13.0 ), 1.5 );
/* 21 */
/* 22 */   printf( "%.0f\n", (*p_f)( 38.0, ergebnis ) );
/* 23 */
/* 24 */   return 0;
}
```

Mit dem `typedef` *in Zeile 3 wird ein neuer Datentyp geschaffen mit dem Namen* `p_t`.
Der neue Typ steht für **Zeiger auf eine Funktion, die zwei** `double`**-Parameter bekommt und eine** `double` **liefert**. (2)
In Zeile 17 wird ein Feld angelegt mit zwei Elementen vom Typ (3)
Zeiger auf... und mit den Adressen der Funktionen `subtrahiere()` (5)
(`arr[0]`*) beziehungsweise* `addiere()` *(*`arr[1]`*) initialisiert.*

In Zeile 18 wird eine Zeigervariable angelegt vom Typ **Zeiger auf** `p_t`*, also* **Zeiger auf Zeiger auf Funktion...** *und mit der Adresse des untersten Feldelements initialisiert, also mit der Adresse von* `arr[0]` *(worin wiederum die Adresse von* `subtrahiere()` *steht).* (5)

In Zeile 20 wird zuerst `arr[0]( 15.5, 13.0 )` *ausgeführt, also* `subtrahiere( 15.5, 13.0 )` *mit dem Ergebnis* `2.5`*.* (4)

`p_f` *wird durch das* `++` *um eins erhäht, zeigt jetzt also nicht mehr auf* `a[0]`*, sondern auf* `a[1]` *(darin steht die Adresse von* `addiere()`*).* (2)

Mit dem Zwischenergebnis `2.5` *und mit* `1.5` *wird das aufgerufen, worauf* `p_f` *nach dem Erhöhen zeigt, also die Funktion* `addiere()`*.* (4)

Das Resultat `4.0` *wird in einer neuen* `double`*-Variablen* `ergebnis` *gespeichert.* (4)

Beim Aufruf von `printf()` *wird neben dem Formatstring das Ergebnis eines weiteren Funktionsaufrufs übergeben, und zwar wird die Funktion aufgerufen, die an der Stelle* `*p_f` *steht, also nochmalls* `addiere()` *mit den Parametern* `38.0` *und* `ergebnis` *(*`4.0`*). Das Resultat* `42` *wird ausgegeben.* (2)

Summe: 31
