Fachhochschule Wedel
Prof. Dr. Uwe Schmidt

5. August 1996

Aufgaben zur Klausur **C** und **Objektorientierte Programmierung** im SS 96 (WI63)

Zeit: 120 Minuten erlaubte Hilfsmittel: keine

Bitte tragen Sie Ihre Antworten und fertigen Lösungen ausschließlich an den gekennzeichneten Stellen in das Aufgabenblatt ein. Ist ihre Lösung wesentlich umfangreicher, so überprüfen Sie bitte nochmals Ihren Lösungsweg.

Bei allen C Routinen sollen die Parameter und Funktionsresultate mit ANSI–C Prototypen angegeben werden.

Viel Erfolg !

Diese Klausur besteht einschließlich dieses Deckblattes aus 13 Seiten

---

Aufgabe 1:

Gegeben sei das folgende C–Programm:

```
#include <stdio.h>

#define maxm(x,y) ((x > y) ? (x) : (y))

long maxf(long x, long y) { return x > y ? x : y; }

unsigned cnt = 0;

#define eval(x) ( ++cnt, (x) )


int main(void) {
    long f[] = { 0, -2, +4, -6, +8 };
    long res;

    double g[] = { 0.5, 2.1, 3.9 };
    double r;

    cnt = 0;
    res = maxm( maxm( eval(f[1]), eval(f[2]) ),
                eval(f[3]) );

    printf(" res = %ld, cnt = %u\n", res, cnt);

    cnt = 0;
    res = maxf( maxf( eval(f[1]), eval(f[2]) ),
                eval(f[3]) );

    printf(" res = %ld, cnt = %u\n", res, cnt);

    cnt = 0;
    r = maxm( eval(g[1]), eval(g[2]) );

    printf(" r = %3.1f, cnt = %u\n", r, cnt);

    cnt = 0;
    r = maxf( eval(g[1]), eval(g[2]) );

    printf(" r = %3.1f, cnt = %u\n", r, cnt);

    return 0;
}
```

Welche vier Ausgabezeilen erzeugt dieses Programm:

1) .....

2) .....

3) .....

4) .....

Aufgabe 2:

Gegeben sei das folgende C–Programmstück für die Verarbeitung von binären Bäumen:

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

typedef char * String;

typedef struct node * Bintree;
struct node {
    String info;
    Bintree l;
    Bintree r;
};

#define empty ((Bintree)0)

Bintree mk1(String e) {
    Bintree t;
    t = malloc(sizeof(*t));
    if ( !t )
        exit(1);
    t->info = e;
    t->l = empty;
    t->r = empty;
    return t;
}

int compare(String e1, String e2) {
    return
        strcmp(e1,e2);
}

extern Bintree insert(String e1, Bintree t);
```

Implementieren Sie die fehlende *insert* Routine. Diese trägt ein Element, hier eine Zeichenreihe, in einen binären Baum ein, falls das Element noch nicht in dem Baum vorhanden ist.

Nutzen Sie die in dem Programmstück vorgegebenen Datentypen, Makros und Funktionen.

(Lösung auf der nächsten Seite).

Die *insert* Routine:

Aufgabe 3:

Gegeben sei das folgende C–Programm zur Verarbeitung von Mengen als Bitstrings.

```
#include <stdio.h>

typedef unsigned char Set;
#define SetMax 8

void printSet(Set s) {
    unsigned int i = SetMax;
    while ( i-- ≠ 0 )
        printf("%1u", (unsigned int)((s ≫ i) & 1));
}

static unsigned int linecnt = 0;
#define PRINT(s) { printf("%2u) ", ++linecnt); printSet(s); printf("\n"); }

#define single(i) ( (Set)(1 ≪ (i)) )
#define first(n) (single(n) - 1)
#define interval(n,m) (first(m+1) ^ first(n))

int main(void) {
    Set s1;

    PRINT( (Set)0 );
    PRINT( (Set)-1 );
    PRINT( single(SetMax / 2) );

    PRINT( 1 | 4 );
    PRINT( 1 || 4 );
    PRINT( first(SetMax / 2) );

    PRINT( interval(0,1) );
    PRINT( interval(2,5) );
    PRINT( interval(0,SetMax) );

    s1 = interval(0,4) & ~interval(2,6); PRINT(s1);
    s1 = interval(0,4) ^ interval(2,6); PRINT(s1);

    s1 = 4 + 16 + 32;
    s1 ^= s1 & (~s1 + 1); PRINT(s1);

    return 0;
}
```

Die Mengen sind in diesem Beispiel 8 Bits lang, können also die Elemente $0, 1, \ldots, 7$ enthalten. *printSet* gibt eine Menge im Binärformat aus. Die Menge, die nur die 1 enthält würde als 00000010 ausgegeben werden. Das *PRINT* Makro gibt jeweils eine Menge pro Zeile aus und numeriert die Zeilen durch.

Welche 12 Ausgabezeilen erzeugt dieses Programm?

1) .....

2) .....

3) .....

4) .....

5) .....

6) .....

7) .....

8) .....

9) .....

10) .....

11) .....

12) .....

Aufgabe 4:

Gegeben sei das folgende C++–Programmstück, in dem eine Klasse *Str* definiert wird. Dieses Programmstück stehe in der Datei *str0.cc*.

```
#include <string.h>
#include <iostream.h>

class Str {

protected:
  char * sptr;

public:
  Str() {
    sptr = "";
  }

  Str(char * s) {
    sptr = s;
  }

  virtual char operator [] (unsigned int i) const {
    return sptr[i];
  }

  virtual unsigned int len() const {
    return strlen(sptr);
  }

  friend ostream & operator << (ostream & o, const Str & s) {
    return o << s.sptr;
  }

  virtual ~Str() { }
};
```

Hier wird ein *string* in einer *member*–Variablen *sptr* gespeichert, es gibt 2 Konstruktoren, einen Destruktor, die Operatorfunktion zum indizierten Zugriff auf ein Zeichen und eine Funktion zur Berechnung der Länge der gespeicherten Zeichenreihe.

Aus dieser Klasse soll eine Klasse *Str1* entwickelt werden, bei der für die gespeicherten Zeichenreihen beim Konstruktoraufruf immer eine Kopie auf der Halde angelegt wird und beim Destruktoraufruf der Speicher für diese Kopie wieder freigegeben wird. Zusätzlich müssen für diese Klasse dann auch der *copy*–Konstruktor und der Zuweisungsoperator explizit implementiert werden.

Die Klasse hat also folgende Gestalt:

```
#include "str0.cc"

class Str1 : public Str {

public:
  Str1();

  Str1(char * s);

  Str1(const Str1 & s);

  Str1 & operator = (const Str1 & s);

  ~Str1();

private:
  void def_sptr(char * s) {
    sptr = new char[strlen(s) + 1];
    if (! sptr)
      exit(1);
    strcpy(sptr, s);
  }

  void undef_sptr() {
    delete [] sptr;
  }
};
```

Zwei Hilfsfunktionen sind hier schon vorgegeben, implementieren Sie die anderen Funktionen, nur die Funktionsrümpfe, die *header* sind vorgegeben. Diese Klasse soll in der Datei *str1.cc* gespeichert sein.

(Lösungen bitte auf der folgenden Seite)

Str1::Str1()

. . . . . .

Str1::Str1(**char** ∗ s)

. . . . . .

. . . . . .

Str1::Str1(**const** Str1 & s)

. . . . . .

. . . . . .

Str1 & Str1::**operator** = (**const** Str1 & s)

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Str1::∼Str1()

. . . . . .

Die Klasse *Str1* soll nochmal erweitert werden zu einer Klasse *Str2*, und zwar so, daß die Längenberechnung nur einmal bei der Konstruktion eines Objekts gemacht wird, und nicht bei jedem Aufruf der Methode *len*. Außerdem soll die Operatorfunktion für den indizierten Zugriff sicher gemacht werden, d.h. bei einem illegalen Index soll das Programm abgebrochen werden.

Die Klasse *Str2* hat also folgende Gestalt:

```
#include "str1.cc"

class Str2 : public Str1 {

protected:
  unsigned int slen;

public:
  Str2();

  Str2(char * s);

  Str2(const Str2 & s);

  Str2 & operator = (const Str2 & s);

  ~Str2();

  char operator [] (unsigned int i) const;

  unsigned int len() const;
};
```

Geben Sie die Funktionsrümpfe der *member*–Funktionen an (auf den folgenden Seiten):

Str2::Str2()

. . . . . .

. . . . . .

Str2::Str2(**char** ∗ s)

. . . . . .

. . . . . .

. . . . . .

Str2::Str2(**const** Str2 & s)

. . . . . .

. . . . . .

. . . . . .

Str2 & Str2::**operator** = (**const** Str2 & s)

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Str2::∼Str2()

. . . . . .

char Str2::**operator** [] (**unsigned int** i) **const**

. . . . . .

. . . . . .

. . . . . .

. . . . . .

unsigned Str2::**int** len() **const**

. . . . . .

. . . . . .

. . . . . .

. . . . . .

In der Klasse *Str* sind der indizierte Zugriff und die Längenfunktion als virtuelle Funktionen deklariert. Warum?

Betrachten Sie dazu das folgende Programmstück:

Str & x = Str2("abc");

... x.len() ...
... x[3] ...

. . . . . .

. . . . . .

. . . . . .