# Tentamen
# EDAF30 Programmering i C++

### 17–01–09, 14:00–19:00

*Hjälpmedel:* En valfri C++-bok. OH-bilderna från föreläsningarna är *inte* tillåtna.

Du ska i dina lösningar visa att du behärskar C++ och även att du kan använda C++ standardklasser och algoritmer. Rena ”C-lösningar” på problem som med fördel kan lösas enligt mera objektorienterade principer, eller egen implementering av sådant som finns i standardbiblioteket, kan därför ge poängavdrag, även om de är korrekta. I bedömningen av en lösning kan även minneshantering beaktas där det är relevant. Om du använder saker ur standardbiblioteket i din kod, var tydlig med att du gör det. Använd fullt kvalificerade namn eller `using`-direktiv för de namn du använder.

Uppgifterna ger preliminärt $10 + 12 + 14 + 14 = 50$ poäng. För godkänt krävs preliminärt 25 poäng (betygsgränser 3/25, 4/33, 5/42). Observera att ordningsföljden eller poängantalet för uppgifterna inte nödvändigtvis avspeglar deras svårighet.

---

1. Vi vill i en klass hålla reda på namn. Namnen lagrar vi i ett objekt av STL-klassen `set`. Vi har valt att i mängden lagra *pekare* på strängarna som innehåller namnen. Klassen ser ut så här:

```
class NameList {
public:
    NameList() {}
    ~NameList() {}
    void insert(const std::string& name) {
        names.insert(new std::string(name));
    }
    void printSorted() const {
        for (list_type::const_iterator it = names.begin(); it != names.end(); ++it) {
            std::cout << *it << std::endl;
        }
    }
private:
    typedef std::set<std::string*> list_type;
    list_type names;
};
```

   a) Klassen innehåller en uppenbar minnesläcka. Förklara varför klassen läcker minne och visa hur klassen ska ändras för att korrigera felet.

   b) Utskriften i `printSorted` blir inte alls som förväntat — man får hexadecimala tal i stället för namn. Varför? Korrigera funktionen så att det skrivs ut namn i stället för tal.

   c) Även efter korrigeringen blir det fel — namnen skrivs inte ut i sorterad ordning, trots att ett `set` ju ska hålla elementen sorterade. Varför inte? Korrigera också detta fel.

2. För full poäng på dessa deluppgifter krävs utförliga och väl motiverade svar.

   a) Nedanstående funktion är korrekt C++, men gör nog inte det man önskar. Förklara vad funktionen gör så som den är skriven här och skriv om den så att faktiskt kollar om $x \in [-0.5, 0.5]$.

```
bool insideLimits(double x) {
        return -0.5 <= x <= 0.5;
}
```

   b) Nedanstående klass innehåller ett fel. Förklara vad som är fel och korrigera felet.
   Dessutom kan en rad i klassen tas bort utan att klassens funktion påverkas. Vilken rad? Varför är den onödig?

```
1   class X {
2   public:
3       X() : v(nullptr) {}
4       X(int n) : v(new int(n)) {}
5       ~X() {
6           if (v != nullptr)
7               delete[] v;
8       }
9   private:
10      int* v; // dynamic array of int's
11  };
```

   c) Betrakta följande klass:

```
class Y {
public:
        Y();
        virtual ~Y();
};
```

   Uttrycket `sizeof(Y)` har värdet 4 (eller 8, kan variera mellan olika datorer). Varför? Klassen innehåller ju inga medlemsvariabler som tar plats, så resultatet borde väl ha blivit 0?

   d) Ibland behöver man inte tilldela eller kopiera objekt, och ibland vill man omöjliggöra att dessa operationer utförs. Hur kan man göra det omöjligt att kopiera objekt av en viss klass?

   e) Vad händer när följande program exekveras?

```
#include<iostream>


int main()
{
    for(unsigned u = 10; u >= 0; --u) {
        std::cout << u << std::endl;
    }

    return 0;
}
```

   Svara med vad som skrivs ut och/eller förklara vad som händer.

3. Objekt av klassen `PositiveInteger` används för att lagra positiva heltal. Om man försöker lagra ett negativt tal får man ett exception. Man har implementerat klassen på följande sätt:

```
class PositiveInteger {
public:
    PositiveInteger(int v) { set(v); }
    int get() const noexcept{ return value; }
    void set(int v) {
        if (v < 0) {
            throw std::logic_error("Value error");
        }
        value = v;
    }
private:
    int value;
};
```

Du ska nu generalisera `PositiveInteger` till `CheckedValue` som ska kunna

1. lagra värden av godtycklig typ
2. utföra en godtycklig kontroll av värdet.

Exempel:

```
void f() {
    CheckedValue<int, pos> p(1); // integers >= 0
    p.set(3);  // ska gå bra
    p.set(-3); // ska ge exception
}

void g() {
    CheckedValue<string, long_string<5> > q("abcdef"); // strings, more than 5 chars
    q.set("qwertyuio"); // ska gå bra
    q.set("abc");       // ska ge exception
}
```

a) Implementera `CheckedValue`, samt `pos` och `long_string` så att exemplen `f()` och `g()` ovan fungerar som beskrivet.

b) Om vi har ett `PositiveInteger`-objekt `p`, så är tilldelningssatsen

```
p = 3;
```

giltig C++ . Förklara vad som händer när den tilldelningssatsen exekveras.

c) Däremot kan man inte kompilera nedanstående rader:

```
PositiveInteger p{3};

int i = p;
```

utan man får följande felmeddelande:
`error: cannot convert 'PositiveInteger' to 'int' in initialization.`
Komplettera klassen `PositiveInteger` så att satsen `int i = p;` fungerar.

4. En klass `Time` som beskriver tidpunkter med timme, minut och sekund ska skrivas. Implementera konstruktioner så att följande program fungerar:

```
#include <iostream>
using std::cout;
using std::endl;

#include "time.h"

int main()
{
    Time t1;              // current local time, e.g. 12:48:01
    Time t2(15,30,1);     // 15:30:01
    Time t3("16:09:27");  // 16:09:27

    t1 += 120;            // two minutes later: 12:50:01
    ++t1;                 // one second later: 12:50:02
    cout << t3 - t2 << endl;    // difference in seconds: 2366
    t3 = t1;              // t3 becomes 12:50:02
    cout << t3 << endl;   // prints 12:50:02
}
```

Notera att utskriften skriver på formatet `TT:MM:SS`, där timmar, minuter och sekunder skrivs med två siffror. Utskriftsrutinen ska alltså lägga till inledande nollor där det behövs.

Du kan förutsätta att alla tider gäller under en bestämd dag och att alla parametrar är korrekta. Till exempel lägger man aldrig till så många sekunder till en tid så att midnatt passeras. Använd funktioner ur standardbiblioteket `<ctime>` (`<time.h>`), se bilaga, för att ta reda på aktuell tid.

Du ska visa hur implementeringen görs med en header-fil och en implementeringsfil med alla `include`-direktiv och andra direktiv som behövs.

```
CTIME(3)         BSD Library Functions Manual             CTIME(3)

NAME
     asctime, asctime_r, ctime, ctime_r, difftime, gmtime, gmtime_r,
     localtime, localtime_r, mktime, timegm -- transform binary date and time
     values

LIBRARY
     Standard C Library (libc, -lc)

SYNOPSIS
     #include <time.h>

     extern char *tzname[2];

     char *
     asctime(const struct tm *timeptr);

     char *
     asctime_r(const struct tm *restrict timeptr, char *restrict buf);

     char *
     ctime(const time_t *clock);

     char *
     ctime_r(const time_t *clock, char *buf);

     double
     difftime(time_t time1, time_t time0);

     struct tm *
     gmtime(const time_t *clock);

     struct tm *
     gmtime_r(const time_t *clock, struct tm *result);

     struct tm *
     localtime(const time_t *clock);

     struct tm *
     localtime_r(const time_t *clock, struct tm *result);

     time_t
     mktime(struct tm *timeptr);

     time_t
     timegm(struct tm *timeptr);

DESCRIPTION
     The functions ctime(), gmtime(), and localtime() all take as an argument
     a time value representing the time in seconds since the Epoch (00:00:00
     UTC, January 1, 1970; see time(3)).

     The function localtime() converts the time value pointed at by clock.  It
     returns a pointer to a ‘‘struct tm’’ (described below), which contains
     the broken-out time information for the value after adjusting for the
     current time zone (and any other factors such as Daylight Saving Time).
     Time zone adjustments are performed as specified by the TZ environment
     variable (see tzset(3)).  The function localtime() uses tzset(3) to ini-
     tialize time conversion information, if tzset(3) has not already been
     called by the process.
```

After filling in the tm structure, localtime() sets the tm_isdst'th element of tzname to a pointer to an ASCII string containing the time zone abbreviation to be used with localtime()'s return value.

The function gmtime() also converts the time value, but makes no time zone adjustment. It returns a pointer to a tm structure (described below).

The ctime() function adjusts the time value for the current time zone, in the same manner as localtime(). It returns a pointer to a 26-character string of the form:

```
Thu Nov 24 18:22:48 1986\n\0
```

All of the fields have constant width.

The ctime_r() function provides the same functionality as ctime(), except that the caller must provide the output buffer buf (which must be at least 26 characters long) to store the result. The localtime_r() and gmtime_r() functions provide the same functionality as localtime() and gmtime(), respectively, except the caller must provide the output buffer result.

The asctime() function converts the broken-out time in the structure tm (pointed at by *timeptr) to the form shown in the example above.

The asctime_r() function provides the same functionality as asctime(), except that the caller provides the output buffer buf (which must be at least 26 characters long) to store the result.

The functions mktime() and timegm() convert the broken-out time (in the structure pointed to by *timeptr) into a time value with the same encoding as that of the values returned by the time(3) function (that is, seconds from the Epoch, UTC). The mktime() function interprets the input structure according to the current timezone setting (see tzset(3)). The timegm() function interprets the input structure as representing Universal Coordinated Time (UTC).

The original values of the tm_wday and tm_yday components of the structure are ignored. The original values of the other components are not restricted to their normal ranges and will be normalized, if need be. For example, October 40 is changed into November 9, a tm_hour of -1 means 1 hour before midnight, tm_mday of 0 means the day preceding the current month, and tm_mon of -2 means 2 months before January of tm_year. (A positive or zero value for tm_isdst causes mktime() to presume initially that summer time (for example, Daylight Saving Time) is or is not (respectively) in effect for the specified time. A negative value for tm_isdst causes the mktime() function to attempt to divine whether summer time is in effect for the specified time. The tm_isdst and tm_gmtoff members are forced to zero by timegm().)

On successful completion, the values of the tm_wday and tm_yday components of the structure are set appropriately, and the other components are set to represent the specified calendar time, but with their values forced to their normal ranges; the final value of tm_mday is not set until tm_mon and tm_year are determined. The mktime() function returns the specified calendar time; if the calendar time cannot be represented, it returns -1;

The difftime() function returns the difference between two calendar times, (time1 - time0), expressed in seconds.

External declarations, as well as the tm structure definition, are contained in the <time.h> include file. The tm structure includes at least the following fields:

```
int tm_sec;     /* seconds (0 - 60) */
int tm_min;     /* minutes (0 - 59) */
int tm_hour;    /* hours (0 - 23) */
int tm_mday;    /* day of month (1 - 31) */
int tm_mon;     /* month of year (0 - 11) */
int tm_year;    /* year - 1900 */
int tm_wday;    /* day of week (Sunday = 0) */
int tm_yday;    /* day of year (0 - 365) */
int tm_isdst;   /* is summer time in effect? */
char *tm_zone;  /* abbreviation of timezone name */
long tm_gmtoff; /* offset from UTC in seconds */
```

The field tm_isdst is non-zero if summer (i.e., Daylight Saving) time is in effect.

The field tm_gmtoff is the offset (in seconds) of the time represented from UTC, with positive values indicating locations east of the Prime Meridian.

## SEE ALSO

date(1), gettimeofday(2), getenv(3), time(3), tzset(3), tzfile(5)

## STANDARDS

The asctime(), ctime(), difftime(), gmtime(), localtime(), and mktime() functions conform to ISO/IEC 9899:1990 (‘‘ISO C90’’), and conform to ISO/IEC 9945-1:1996 (‘‘POSIX.1’’) provided the selected local timezone does not contain a leap-second table (see zic(8)).

The asctime_r(), ctime_r(), gmtime_r(), and localtime_r() functions are expected to conform to ISO/IEC 9945-1:1996 (‘‘POSIX.1’’) (again provided the selected local timezone does not contain a leap-second table).

The timegm() function is not specified by any standard; its function cannot be completely emulated using the standard functions described above.

## HISTORY

This manual page is derived from the time package contributed to Berkeley by Arthur Olson and which appeared in 4.3BSD.

## BUGS

Except for difftime(), mktime(), and the _r() variants of the other functions, these functions leaves their result in an internal static object and return a pointer to that object. Subsequent calls to these function will modify the same object.

The C Standard provides no mechanism for a program to modify its current local timezone setting, and the POSIX-standard method is not reentrant. (However, thread-safe implementations are provided in the POSIX threaded environment.)

The tm_zone field of a returned tm structure points to a static array of characters, which will also be overwritten by any subsequent calls (as well as by subsequent calls to tzset(3) and tzsetwall(3)).

Use of the external variable tzname is discouraged; the tm_zone entry in the tm structure is preferred.

```
TIME(3)          BSD Library Functions Manual              TIME(3)

NAME
     time -- get time of day

LIBRARY
     Standard C Library (libc, -lc)

SYNOPSIS
     #include <time.h>

     time_t
     time(time_t *tloc);

DESCRIPTION
     The time() function returns the value of time in seconds since 0 hours, 0
     minutes, 0 seconds, January 1, 1970, Coordinated Universal Time, without
     including leap seconds.  If an error occurs, time() returns the value
     (time_t)-1.

     The return value is also stored in *tloc, provided that tloc is non-null.

ERRORS
     The time() function may fail for any of the reasons described in
     gettimeofday(2).

SEE ALSO
     gettimeofday(2), ctime(3)

STANDARDS
     The time function conforms to IEEE Std 1003.1-2001 (‘‘POSIX.1’’).

BUGS
     Neither ISO/IEC 9899:1999 (‘‘ISO C99’’) nor IEEE Std 1003.1-2001
     (‘‘POSIX.1’’) requires time() to set errno on failure; thus, it is impos-
     sible for an application to distinguish the valid time value -1 (repre-
     senting the last UTC second of 1969) from the error return value.

     Systems conforming to earlier versions of the C and POSIX standards
     (including older versions of FreeBSD) did not set *tloc in the error
     case.

HISTORY
     A time() function appeared in Version 6 AT&T UNIX.

BSD              July 18, 2003                BSD
```