Tentamen

KURSNAMN	**Programutveckling**
PROGRAM: namn **åk / läsperiod**	Åk 1 på DAI/EI samt Åk 1/2 på MEI
KURSBETECKNING	LEU480 / LEU481
EXAMINATOR	Peter Lundin
TID FÖR TENTAMEN	Måndagen den 2015-08-17, kl 14.00–18.00
HJÄLPMEDEL	Inga hjälpmedel tillåtna utöver bilagor i tentamenstesen.
ASNV. LÄRARE namn **telnr** **besöker tentamen kl**	Pierre Kleberger 772 5225 Ca 15.15, samt 16.45
OBS ! **Tid och plats för granskning**	Annonseras på kurshemsidan
ÖVRIG INFORMATION	Betygsgränser : Max 50 poäng **3:** 20–30, **4:** 31–40, **5:** 41–50

Anvisningar

Vid lösningar av tentamen gäller följande generella anvisningar:

- Om inget annat anges får man använda listade standardfunktioner i bifogat blad "C-reference card".
- Om inget annat anges så behövs ingen indatakontroll göras.
- Inkluderingsfiler (ex. **#include <....h>**) behöver ej anges.
- Använda variabler skall vara deklarerade.
- Programmens syntax behöver ej vara fullt korrekt för full poäng.
- För full poäng krävs en i princip fungerande, läsbar, indenterad och "begriplig" lösning. Onödigt krångliga lösningar kan ge poängavdrag.

Lycka till!

Uppgifter

1. Förklara kortfattat och förtydliga med ett exempel följande begrepp: (3p)
   (a) Funktionsdeklaration
   (b) Fält
   (c) Pekare

2. Studera programmet i Bilaga A. Vad skriver programmet ut på skärmen? (3p)

3. Följande enum är deklarerad:

```
1  enum DAG { MON, TIS, ONS, TOR, FRE, LOR, SON };
```

   Skriv funktionen `imorgon` som returnerar den dag som kommer imorgon.
   Funktionsprototypen är:
   `enum DAG imorgon(enum DAG idag)` (3p)

4. Volymen av en kon beräknas med formeln:

$$V = \frac{\pi r^2 h}{3}$$

där r är konens bas-radie och h är konens höjd. Skriv en funktion som givet inparametrarna r och h beräknar konens volym. Antag att π har definierats med följande kod:

```
1  #define M_PI 3.14159265359
```

(3p)

Tips: Tänk på datatyperna.

5. Skriv en funktion som kvarderar (dvs. multiplicerar med sig själv) varje element i ett fält. Funktionen skall returnera antalet element som förändrats; om ett element hade värdet 0 blir $0^2 = 0$ och har således inte förändrats.

   Funktionsdeklarationen är:
   `int my_square(int v[], int langd)` (4p)

   Ex:

```
1  int v[] = { 5, 5, 3, 0, 4, 0 };
2  int v_len = sizeof(v) / sizeof(int);
3  int resultat = my_square(v, v_len);
```

   skulle resultera i värdena $\{25, 25, 9, 0, 16, 0\}$ i fältet `v[]` och värdet 4 i variablen `resultat`.

6. Skriv ett program som genererar slumptal och sparar dessa i en text-fil. Antalet slumptal som skall genereras och dess intervall (dvs. största och minsta slumptal) besvaras av användaren. (Felhantering för öppnande och skrivning till fil behöver inte tas omhand.) (6p)

   Körexempel:

```
Minsta slumptal: 10
Storsta slumptal: 20
Antal slumptal: 5
Ange Filnamn: slumptal.txt
```

   Innehållet i filen `slumptal.txt` efter att programmet körts:

```
1 18
2 18
3 18
4 13
5 10
```

7. Skriv en funktion som beräknar antalet bokstäver och siffror i en textsträng. Funktionen skall ta strängen som inparameter och returnera antalet bokstäver och antalet siffror via parametrar. ASCII-värdena för 'A'–'Z' är 65–90, 'a'–'z' är 97–122, och '0'–'9' är 48–57. Övriga tecken behöver inte tas hänsyn till. (6p)

   Ex:
   Följande text har 23 bokstäver och 1 siffror:

   ```
   En kort rad text med 1 siffra i.
   ```

8. Antag att vi skall bygga ett litet telefonregister med kortnummer som används för att ringa till kollegor inom ett företag. Följande post-typer är definierade:

```
1  #define MAX 255
2  #define NAMNMAX 32
3
4  struct Person {
5      char namn[NAMNMAX];
6      short kortnummer;
7  };
8
9  struct Register {
10     struct Person personer[MAX];
11     int antal;
12 };
```

   (a) Skriv en funktion som givet ett telefonregister och ett namn (sträng) returnerar kortnummret till den givna personen. Om namnet inte finns, returneras -1. (5p)

   (b) Skriv en funktion som givet ett telefonregister och en person, lägger till denna i telefonregistret. Funktionen returnerar 0 om det fanns plats för personen, annars -1. (6p)

   Tips: Det är tillåtet att använda standardfunktionen:
   `int strcmp(char *s1, char *s2)`
   Funktionen returnerar:

   - ett negativt värde om `s1` kommer före `s2` (i lexikografisk ordning, d.v.s. i ett lexikon),
   - ett positivt värde om `s1` kommer efter `s2`,
   - 0 om strängarna är lika.

9. Skriv en funktion, samt dess hjälpfunktion, som givet ett fält av tal, skriver ut ett histogram med freqvensen av varje tal. Histogrammet skall skrivas ut i *stigande* ordning. Ordningen på värdena i det givna fältet får förändras, därför tar vi hjälp av en sorterings-funktion som sorterar fältet innan vi skriver ut historgrammet.

   Studera följande programkod:

```
1  void histogram(int v[], int len);
2  void sortera(int v[], int len);
3  void skriv_pelare(int varde, int antal);
4
5  int main()
6  {
7    int v[] = { 5, 5, 3, 0, 4, 0 };
8    int v_len = sizeof(v) / sizeof(int);
9    histogram(v, v_len);
10   return 0;
11 }
12
13 void histogram(int v[], int v_len)
14 {
15   // Deklarationer
16   ...
17
18   // Sortera faltet
19   sortera(v, v_len);
20
21   // Implementera utskriften av histogrammet
22   ...
23   }
24
25 void sortera(int v[], int v_len)
26   {
27   // Implementationen av sortering av faltet
28   ...
29 }
30
31 void skriv_pelare(int varde, int freqvens)
32 {
33   // Skriver ut en pelare, dvs. freqvensen for det givna
       vardet.
34   printf("%i ", varde);
35   for(int i = 0; i < antal; i++) printf("X");
36   printf("\n");
37 }
```

En utskrift från programmet ser ut enligt följande:

```
0 XX
3 X
4 X
5 XX
```

Din uppgift är att implementera följande funktioner:

(a) Implementera `void sortera(int v[], int v_len)`. Funktionen skall sortera innehållet i det givna fältet (i stigande ordning). (6p)

(b) Fortsätt implementera `void histogram(int v[], int v_len)`. Funktionen skall skriva ut histogrammet i stigande ordningen (med hjälp av de andra hjälpfunktionerna). (5p)

Notera: Varje deluppgift kan lösas var för sig (även om den andra deluppgift inte löses).

A Programkod till Uppgift 2

```
1  #include <stdio.h>
2
3  #define MAX 5
4
5  int main()
6  {
7      int j;
8      for(int i = 1; i <= MAX; i++) {
9          j = i;
10         while (j > 0) printf("%i", j--);
11         printf("\n");
12     }
13     return 0;
14 }
```

C Reference Card (ANSI)

Program Structure/Functions

type fnc(*type*$_1$,...)	function declarations
type name	external variable declarations
`main() {`	main routine
declarations	local variable declarations
statements	
`}`	
type fnc(*arg*$_1$,...) {	function definition
declarations	local variable declarations
statements	
`return` *value*`;`	
`}`	
`/* */`	comments
`main(int argc, char *argv[])`	main with args
`exit(`*arg*`)`	terminate execution

C Preprocessor

include library file	`#include <`*filename*`>`
include user file	`#include "`*filename*`"`
replacement text	`#define` *name text*
replacement macro	`#define` *name*`(`*var*`)` *text*
Example. `#define max(A,B) ((A)>(B) ? (A) : (B))`	
undefine	`#undef` *name*
quoted string in replace	`#`
concatenate args and rescan	`##`
conditional execution	`#if, #else, #elif, #endif`
is *name* defined, not defined?	`#ifdef, #ifndef`
name defined?	`defined(`*name*`)`
line continuation char	`\`

Data Types/Declarations

character (1 byte)	`char`
integer	`int`
float (single precision)	`float`
float (double precision)	`double`
short (16 bit integer)	`short`
long (32 bit integer)	`long`
positive and negative	`signed`
only positive	`unsigned`
pointer to `int`, `float`,...	`*int, *float,...`
enumeration constant	`enum`
constant (unchanging) value	`const`
declare external variable	`extern`
register variable	`register`
local to source file	`static`
no value	`void`
structure	`struct`
create name by data type	`typedef` *typename*
size of an object (type is `size_t`)	`sizeof` *object*
size of a data type (type is `size_t`)	`sizeof(`*type name*`)`

Initialization

initialize variable	*type name*`=`*value*
initialize array	*type name*`[]={`*value*$_1$,...`}`
initialize char string	`char` *name*`[]="`*string*`"`

Constants

long (suffix)	`L` or `l`
float (suffix)	`F` or `f`
exponential form	`e`
octal (prefix zero)	`0`
hexadecimal (prefix zero-ex)	`0x` or `0X`
character constant (char, octal, hex)	`'a', '\`*ooo*`', '\x`*hh*`'`
newline, cr, tab, backspace	`\n, \r, \t, \b`
special characters	`\\, \?, \', \"`
string constant (ends with `'\0'`)	`"abc...de"`

Pointers, Arrays & Structures

declare pointer to *type*	*type* `*`*name*
declare function returning pointer to *type*	*type* `*f()`
declare pointer to function returning *type*	*type* `(*pf)()`
generic pointer type	`void *`
null pointer	`NULL`
object pointed to by *pointer*	`*`*pointer*
address of object *name*	`&`*name*
array	*name*`[`*dim*`]`
multi-dim array	*name*`[`*dim*$_1$`][`*dim*$_2$`]...`

Structures

`struct` *tag* `{`	structure template
declarations	declaration of members
`};`	

create structure	`struct` *tag name*
member of structure from template	*name*`.`*member*
member of pointed to structure	*pointer* `->` *member*
Example. `(*p).x` and `p->x` are the same	
single value, multiple type structure	`union`
bit field with *b* bits	*member* `:` *b*

Operators (grouped by precedence)

structure member operator	*name*`.`*member*
structure pointer	*pointer*`->`*member*
increment, decrement	`++, --`
plus, minus, logical not, bitwise not	`+, -, !, ~`
indirection via pointer, address of object	`*`*pointer*`, &`*name*
cast expression to type	`(`*type*`)` *expr*
size of an object	`sizeof`
multiply, divide, modulus (remainder)	`*, /, %`
add, subtract	`+, -`
left, right shift [bit ops]	`<<, >>`
comparisons	`>, >=, <, <=`
comparisons	`==, !=`
bitwise and	`&`
bitwise exclusive or	`^`
bitwise or (incl)	`|`
logical and	`&&`
logical or	`||`
conditional expression	*expr*$_1$ `?` *expr*$_2$ `:` *expr*$_3$
assignment operators	`+=, -=, *=, ...`
expression evaluation separator	`,`

Unary operators, conditional expression and assignment operators group right to left; all others group left to right.

Flow of Control

statement terminator	`;`
block delimeters	`{ }`
exit from `switch`, `while`, `do`, `for`	`break`
next iteration of `while`, `do`, `for`	`continue`
go to	`goto` *label*
label	*label*`:`
return value from function	`return` *expr*

Flow Constructions

`if` statement	`if (`*expr*`)` *statement* `else if (`*expr*`)` *statement* `else` *statement*
`while` statement	`while (`*expr*`)` *statement*
`for` statement	`for (`*expr*$_1$`;` *expr*$_2$`;` *expr*$_3$`)` *statement*
`do` statement	`do` *statement* `while(`*expr*`);`

`switch` statement	`switch (`*expr*`) {`
	`case` *const*$_1$`:` *statement*$_1$ `break;`
	`case` *const*$_2$`:` *statement*$_2$ `break;`
	`default:` *statement*
	`}`

ANSI Standard Libraries

`<assert.h>`	`<ctype.h>`	`<errno.h>`	`<float.h>`	`<limits.h>`
`<locale.h>`	`<math.h>`	`<setjmp.h>`	`<signal.h>`	`<stdarg.h>`
`<stddef.h>`	`<stdio.h>`	`<stdlib.h>`	`<string.h>`	`<time.h>`

Character Class Tests `<ctype.h>`

alphanumeric?	`isalnum(c)`
alphabetic?	`isalpha(c)`
control character?	`iscntrl(c)`
decimal digit?	`isdigit(c)`
printing character (not incl space)?	`isgraph(c)`
lower case letter?	`islower(c)`
printing character (incl space)?	`isprint(c)`
printing char except space, letter, digit?	`ispunct(c)`
space, formfeed, newline, cr, tab, vtab?	`isspace(c)`
upper case letter?	`isupper(c)`
hexadecimal digit?	`isxdigit(c)`
convert to lower case?	`tolower(c)`
convert to upper case?	`toupper(c)`

String Operations `<string.h>`

`s`,`t` are strings, `cs`,`ct` are constant strings

length of `s`	`strlen(s)`
copy `ct` to `s`	`strcpy(s,ct)`
up to `n` chars	`strncpy(s,ct,n)`
concatenate `ct` after `s`	`strcat(s,ct)`
up to `n` chars	`strncat(s,ct,n)`
compare `cs` to `ct`	`strcmp(cs,ct)`
only first `n` chars	`strncmp(cs,ct,n)`
pointer to first `c` in `cs`	`strchr(cs,c)`
pointer to last `c` in `cs`	`strrchr(cs,c)`
copy `n` chars from `ct` to `s`	`memcpy(s,ct,n)`
copy `n` chars from `ct` to `s` (may overlap)	`memmove(s,ct,n)`
compare `n` chars of `cs` with `ct`	`memcmp(cs,ct,n)`
pointer to first `c` in first `n` chars of `cs`	`memchr(cs,c,n)`
put `c` into first `n` chars of `cs`	`memset(s,c,n)`

C Reference Card (ANSI)

Input/Output <stdio.h>

Standard I/O

standard input stream	`stdin`
standard output stream	`stdout`
standard error stream	`stderr`
end of file	`EOF`
get a character	`getchar()`
print a character	`putchar(`*chr*`)`
print formatted data	`printf("`*format*`",`arg_1`,...)`
print to string `s`	`sprintf(s,"`*format*`",`arg_1`,...)`
read formatted data	`scanf("`*format*`",&`$name_1$`,...)`
read from string `s`	`sscanf(s,"`*format*`",&`$name_1$`,...)`
read line to string `s` (< `max` chars)	`gets(s,max)`
print string `s`	`puts(s)`

File I/O

declare file pointer	`FILE *`*fp*
pointer to named file	`fopen("`*name*`","`*mode*`")`

modes: `r` (read), `w` (write), `a` (append)

get a character	`getc(`*fp*`)`
write a character	`putc(`*chr*`,`*fp*`)`
write to file	`fprintf(`*fp*`,"`*format*`",`arg_1`,...)`
read from file	`fscanf(`*fp*`,"`*format*`",`arg_1`,...)`
close file	`fclose(`*fp*`)`
non-zero if error	`ferror(`*fp*`)`
non-zero if EOF	`feof(`*fp*`)`
read line to string `s` (< `max` chars)	`fgets(s,max,`*fp*`)`
write string `s`	`fputs(s,`*fp*`)`

Codes for Formatted I/O: `"%-+ 0`*w*`.`*p*`m`*c*`"`

`-`	left justify
`+`	print with sign
space	print space if no sign
`0`	pad with leading zeros
w	min field width
p	precision
m	conversion character: `h` short, `l` long, `L` long double
c	conversion character:
`d,i`	integer
`u`	unsigned
`c`	single char
`s`	char string
`f`	double
`e,E`	exponential
`o`	octal
`x,X`	hexadecimal
`p`	pointer
`n`	number of chars written
`g,G`	same as `f` or `e,E` depending on exponent

Variable Argument Lists <stdarg.h>

declaration of pointer to arguments	`va_list` *name*`;`
initialization of argument pointer	`va_start(`*name*`,`*lastarg*`)`

lastarg is last named parameter of the function

access next unamed arg, update pointer	`va_arg(`*name*`,`*type*`)`
call before exiting function	`va_end(`*name*`)`

Standard Utility Functions <stdlib.h>

absolute value of `int n`	`abs(n)`
absolute value of `long n`	`labs(n)`
quotient and remainder of `ints n,d`	`div(n,d)`
returns structure with `div_t.quot` and `div_t.rem`	
quotient and remainder of `longs n,d`	`ldiv(n,d)`
returns structure with `ldiv_t.quot` and `ldiv_t.rem`	
pseudo-random integer `[0,RAND_MAX]`	`rand()`
set random seed to `n`	`srand(n)`
terminate program execution	`exit(status)`
pass string `s` to system for execution	`system(s)`

Conversions

convert string `s` to double	`atof(s)`
convert string `s` to integer	`atoi(s)`
convert string `s` to long	`atol(s)`
convert prefix of `s` to `double`	`strtod(s,endp)`
convert prefix of `s` (base `b`) to `long`	`strtol(s,endp,b)`
same, but `unsigned long`	`strtoul(s,endp,b)`

Storage Allocation

allocate storage	`malloc(size), calloc(nobj,size)`
change size of object	`realloc(pts,size)`
deallocate space	`free(ptr)`

Array Functions

search `array` for `key`	`bsearch(key,array,n,size,cmp())`
sort `array` ascending order	`qsort(array,n,size,cmp())`

Time and Date Functions <time.h>

processor time used by program	`clock()`
Example. `clock()/CLOCKS_PER_SEC` is time in seconds	
current calendar time	`time()`
$\texttt{time}_2\texttt{-time}_1$ in seconds (`double`)	`difftime(`$\texttt{time}_2$`,`$\texttt{time}_1$`)`
arithmetic types representing times	`clock_t,time_t`
structure type for calendar time comps	`tm`

`tm_sec`	seconds after minute
`tm_min`	minutes after hour
`tm_hour`	hours since midnight
`tm_mday`	day of month
`tm_mon`	months since January
`tm_year`	years since 1900
`tm_wday`	days since Sunday
`tm_yday`	days since January 1
`tm_isdst`	Daylight Savings Time flag

convert local time to calendar time	`mktime(tp)`
convert time in `tp` to string	`asctime(tp)`
convert calendar time in `tp` to local time	`ctime(tp)`
convert calendar time to GMT	`gmtime(tp)`
convert calendar time to local time	`localtime(tp)`
format date and time info	`strftime(s,smax,"`*format*`",tp)`

`tp` is a pointer to a structure of type `tm`

Mathematical Functions <math.h>

Arguments and returned values are `double`

trig functions	`sin(x), cos(x), tan(x)`
inverse trig functions	`asin(x), acos(x), atan(x)`
arctan(y/x)	`atan2(y,x)`
hyperbolic trig functions	`sinh(x), cosh(x), tanh(x)`
exponentials & logs	`exp(x), log(x), log10(x)`
exponentials & logs (2 power)	`ldexp(x,n), frexp(x,*e)`
division & remainder	`modf(x,*ip), fmod(x,y)`
powers	`pow(x,y), sqrt(x)`
rounding	`ceil(x), floor(x), fabs(x)`

Integer Type Limits <limits.h>

The numbers given in parentheses are typical values for the constants on a 32-bit Unix system.

`CHAR_BIT`	bits in `char`	(8)
`CHAR_MAX`	max value of `char`	(127 or 255)
`CHAR_MIN`	min value of `char`	(−128 or 0)
`INT_MAX`	max value of `int`	(+32,767)
`INT_MIN`	min value of `int`	(−32,768)
`LONG_MAX`	max value of `long`	(+2,147,483,647)
`LONG_MIN`	min value of `long`	(−2,147,483,648)
`SCHAR_MAX`	max value of `signed char`	(+127)
`SCHAR_MIN`	min value of `signed char`	(−128)
`SHRT_MAX`	max value of `short`	(+32,767)
`SHRT_MIN`	min value of `short`	(−32,768)
`UCHAR_MAX`	max value of `unsigned char`	(255)
`UINT_MAX`	max value of `unsigned int`	(65,535)
`ULONG_MAX`	max value of `unsigned long`	(4,294,967,295)
`USHRT_MAX`	max value of `unsigned short`	(65,536)

Float Type Limits <float.h>

`FLT_RADIX`	radix of exponent rep	(2)
`FLT_ROUNDS`	floating point rounding mode	
`FLT_DIG`	decimal digits of precision	(6)
`FLT_EPSILON`	smallest x so $1.0 + x \neq 1.0$	(10^{-5})
`FLT_MANT_DIG`	number of digits in mantissa	
`FLT_MAX`	maximum floating point number	(10^{37})
`FLT_MAX_EXP`	maximum exponent	
`FLT_MIN`	minimum floating point number	(10^{-37})
`FLT_MIN_EXP`	minimum exponent	
`DBL_DIG`	decimal digits of precision	(10)
`DBL_EPSILON`	smallest x so $1.0 + x \neq 1.0$	(10^{-9})
`DBL_MANT_DIG`	number of digits in mantissa	
`DBL_MAX`	max `double` floating point number	(10^{37})
`DBL_MAX_EXP`	maximum exponent	
`DBL_MIN`	min `double` floating point number	(10^{-37})
`DBL_MIN_EXP`	minimum exponent	

Send comments and corrections to J.H. Silverman, Math. Dept., Brown Univ., Providence, RI 02912 USA. ⟨jhs@math.brown.edu⟩