Vietnamese-German University

Wing A and Wing B, Block 9, Eastern
International University, Le Lai Street,
Binh Duong New City, Binh Duong Province

Vietnam

Electrical Engineering and Information Technology, B.Eng.

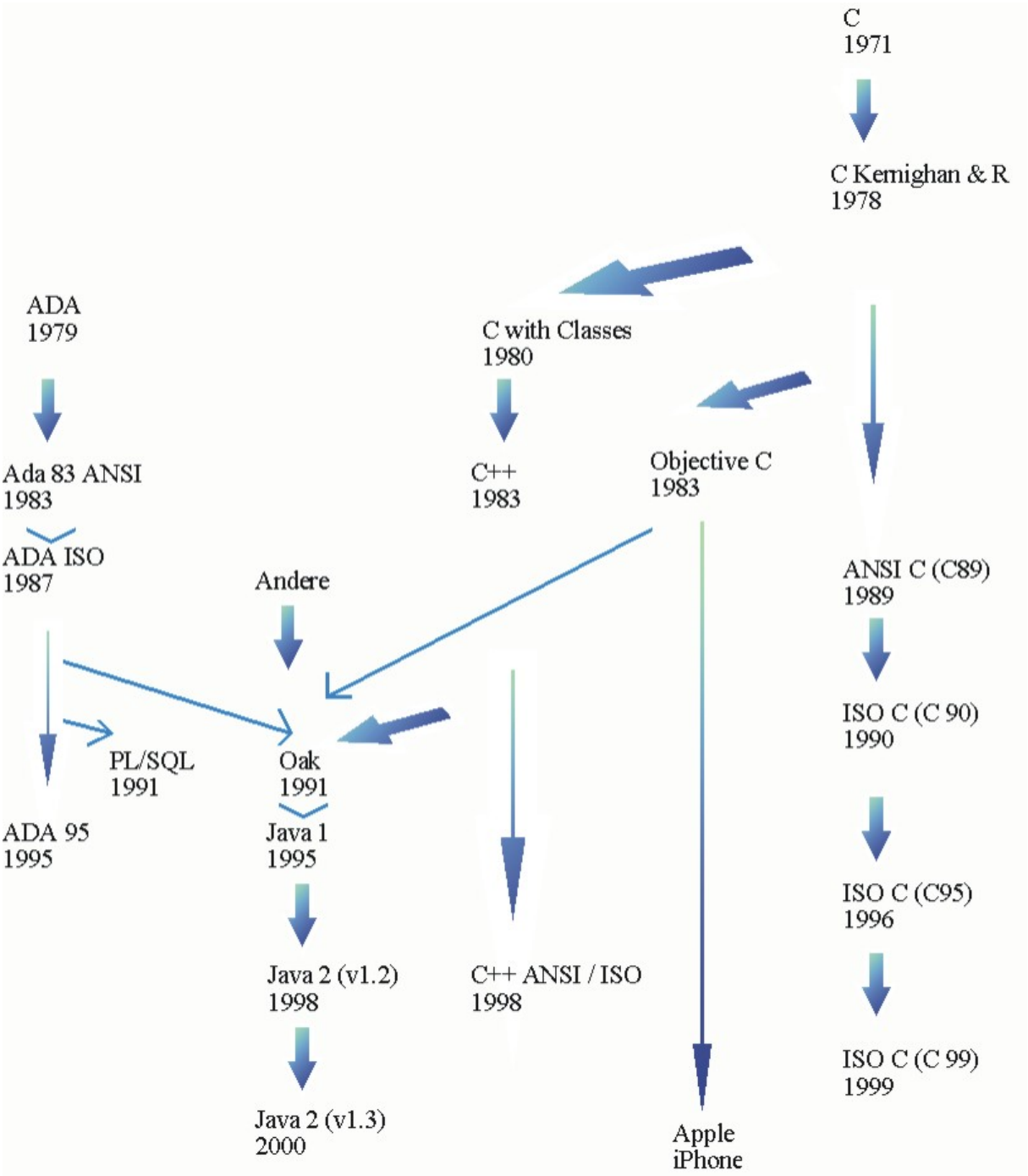
Introduction to the C Programming Language

Script

Bernd Güsmann

1 Information about the C Programming Language

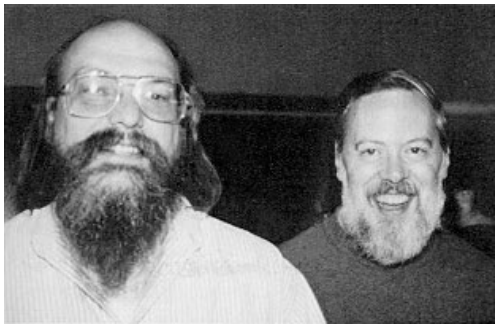
The importance of 'C' is obvious by the following graphic:



Pic. 1-1: Out of the pedigree of programming languages

'C' has been designed by Dennis Ritchie to write the UNIX operating system on the DEC PDP-11. 95% of UNIX are written in 'C'. Many ideas for 'C' are coming from BCPL. A forerunner for BCPL was the language 'B' designed by Ken Thompson, one of the UNIX developers.

Quelle:



http://de.wikipedia.org/wiki/Dennis_Ritchie
Die Bilder sind gemeinfrei

2012 two asteroids were named 294727 Dennisritchie und 300909 Kenthompson.
The asteroids are moving in an asteroid belt between Mars and Jupiter around the sun.

http://astronomy.activeboard.com/t48160002/asteroid-300909-kenthompson/?page=1&sort=oldestFirst

2 Types, Operators and Expressions

2.1 Constants

Integer constants can also be written in octal or hexadecimal form. A leading 0 (zero) on an int constant defines it as an octal constant, a leading 0x or 0X defines it as an hexadecimal constant. Octal constants are interpreted to the base 8, hexadecimal constants are interpreted to the base 16.

The digits for the hexadecimal system are

0, 1, … , 9, A, B, C, D, E, F

E.2-1

decimal	octal	hexadecimal
31	037	0x1F

A character constant like 'x' or '0' is stored in a Byte and has a numerical representation. For example , in the ASCII character set the character zero , or '0', is 48, Which is 0x30.

Have a look into the ASCII table in the appendix.

```
char c;

     c = 120;
     printf("c = %c\n", c);
```

would result in printing 'x' onto the monitor. Of course

```
     c = 'x';
```

gives a quite better understanding of what happens.

A string constant is a sequence of zero or more characters surrounded by double quotes, e.g. "hello, world". The null character \0 is automatically placed at the end of each string. The function

strlen(charstring);

from the standard library returns the length of the character string charstring excluding the final \0-character. This library function needs a

#include <string.h>

in the source file header.

E.2-2 Our version of strlen():

```
int  strlen(char chararray[])
/*****************************************/
/*                                       */
/* strlen() returns the length of chararray.*/
/*                                       */
/*****************************************/
{
int  i;

     i = 0;
     while (chararray[i] != '\0') {
          i++;
     }
     return i;
} /* END_strlen() */
```

calling e.g. by

```
               result = strlen("hello, world");
```

or by

```
     char chararray[4];
          chararray[0] = 'A';
          chararray[1] = 'B';
          chararray[2] = 'C';
          chararray[3] = '\0';
          result = strlen(chararray);
```

Another example of constants are enumeration constants. The type is an enumeration type. The range of values is given with the type definition. Example:

```
     enum boolean {FALSE, TRUE};
     enum boolean alpha;

          alpha = TRUE;
          if (alpha == TRUE){
               printf("True\n");
          }
```

The first constant has the internal representation 0 (in our example this is FALSE). The second constant has the internal representation 1 and so on. This is for information only, not to make use of it.

Another example for the enumeration type is:

```
     enum month {JAN, FEB, MAR, APR, MAI, JUN, JUL, AUG, SEP,
                 OKT, NOV, DEZ};
     enum month  thismonth;

          thismonth = MAR;
```

2.2 Arithmetic Operators

Integer division truncates any fractional part. The expression

```
x % y
```

delivers the remainder of x/y.

E.2-3

A year is a leap year if it is divisible by 4 but not by100, except that a year which is divisible by 400 is nevertheless a leap year.

```
if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0){
    printf("%d is a leap year\n", year);
}
else {
    printf("%d is not a leap year\n", year);
}
```

2.3 Logical Operator

We know the character '!' from the comparison

```
if (x != 3) { ……..... }
```

With '!' we can also invert the truth value of a relational expression:

```
if (!(k < 4)) { .......... }
```

which means it is not true, that k < 4. In other words this is identical to `if (k >= 4) {}`

2.4 Type Conversion

When operands of different types appear in expressions, there is a type conversion. E.g. if one operand is of type float and the other is of type int, then the int operand is converted to float.

Char's and int's may be mixed in arithmetic and relational expressions. The char's are handled as int's.

E.2-4

```
int  atoi(char digits[])
/*******************************************/
/*                                         */
/* Ascii to Integer                        */
/* digits are converted to int.            */
/*                                         */
/*******************************************/
{
int  i, number;

     number = 0;
     i      = 0;
     while ('0' <= digits[i] && digits[i] <= '9') {
          number = number * 10 + (digits[i] - '0');
          i++;
     }
     return number;
} /* END_atoi() */
```

E.2-5

```
char lower(char c)
/*****************************************************/
/*                                                   */
/* converting upper case letter to lower case letter.*/
/*                                                   */
/*****************************************************/
{
     if ('A' <= c && c <= 'Z') {
          return (c + 'a' - 'A');
     }
     else {
          return c;
     }

} /* END_lower() */
```

An explicit type conversion can be forced with a cast. In the construction

```
(type_name) expression
```

the expression is converted to the named type.

E.2-6

```
     float     result;

          result = 3/7;
```

The integer-division on the right side results to integer 0, this is converted to float 0.

```
result = (float)3/7;
```

3 is converted to float, the 2. operand is converted to float, a float division is performed and the result 0.42857 is stored in the left variable. Another way to force float division is

```
result = 3.0/7.0;
```

The parameter declaration in a function prototype declaration forces a type cast if the actual parameter at the time of the function call has another type. The library function sqrt() e.g. is in <math.h> declared to expect a double argument. The call

```
root2 = sqrt(2);
```

forces a cast from 2 to the double-type 2.0 before the algorithm computing the square root starts.

2.5 Increment and Decrement Operators

We give just one example to remember how it works.

E.2-7

```
void strcat(char s[], t_on_s[])
/************************************************/
/*                                              */
/* concatenate t_on_s[] to the end of s[].      */
/* s[] is assumed to have enough space to hold the*/
/* combination.                                 */
/*                                              */
/************************************************/
{
int  i, j;

     i = 0;
     j = 0;
     while (s[i] != '\0') {  /* find end of s. */
          i++;
     }
     while (t_on_s[j] != '\0') {
          s[i++] = t_on_s[j++];
     }
     s[i] = '\0';

} /* END_strcat() */
```

Remember the difference between `k = --n;` and `k = n--;` .

2.6 Bitwise Operators

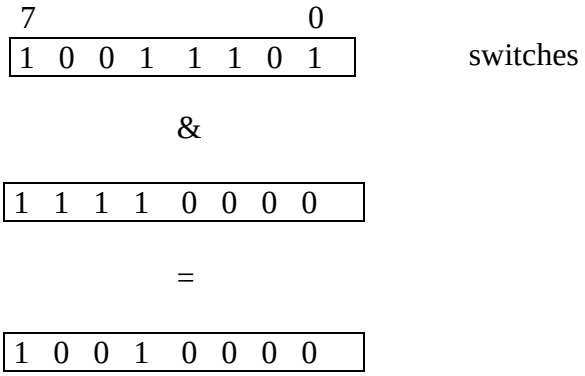
'C' provides a number of operators for bit manipulation. These apply to int-operands and char-operands.

'&' this operator achieves a bitwise AND.

E.2-8

```
unsigned char   switches;

     switches = switches & 0xF0;
```

Bits are count from right to left starting with 0. Here the bits no. 0 to no. 3 of the variable switches are set to 0.

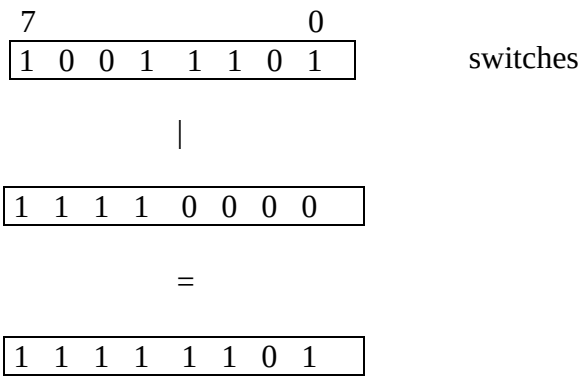


'|' this operator achieves a bitwise OR.
'^' this operator achieves a bitwise exclusive-OR

E.2-9

```
unsigned char  switches;

     switches = switches | 0xF0;
```

Here the bits no. 4 to no. 7 of the variable switches are set to 1.



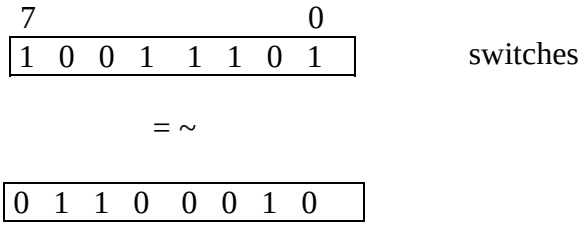
'~' (tilde) this operator achieves a one's complement.

E.2-10

a)

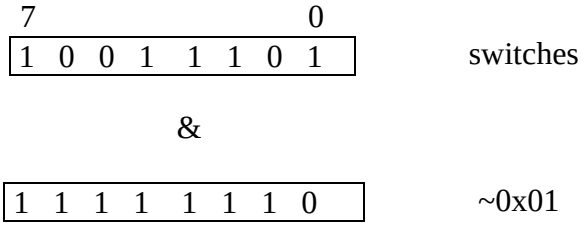
```
unsigned char  switches;

     switches = ~switches;
```

Here the bits no. 0 to no. 7 are toggled.



b) Setting bit no. 0 to 0 :

```
unsigned char  switches;

     switches = switches & ~0x01;
```



The shift operators "<<" and ">>" perform left and right shifts of their left operand by the number of bit positions given by the right operand.

`c << 2` shifts c left by two positions, filling vacated bits with 0

Right shifting of an unsigned variable fills vacated bits with 0.

E.2-11

```
int  bitcount(unsigned char c)
/*************************************************/
/*                                               */
/* counting the number of 1 bits in c.           */
/*                                               */
/*************************************************/
{
int  i;
```

```
     i = 0;
     while (c != 0) {
          if ((c & 0x01) != 0) {
               i++;
          }
          c = c >> 1; /* c is filled with 0's on the left.*/
     }
     return i;
     /* the actual parameter is unchanged.             */
     /* the value of c is copied to the stack and the  */
     /* value on the stack is changed.                 */
} /* END_bitcount() */
```

E.2-12

For a better reading of programs we can define bit masks and use them with bitwise operators. We write the definitions into a header file:

```
/*********************************************************/
/*                                                       */
/*  file : bits.h                                        */
/*  version 1.0                                          */
/*  (c) 2010, B.Guesmann, VGU                            */
/*  Date: 03.05.2010                                     */
/*                                                       */
/*  Contains bit masks                                   */
/*                                                       */
/*********************************************************/

#define BIT0        0x01
#define BIT1        0x02
#define BIT2        0x04
#define BIT3        0x08
#define BIT4        0x10
#define BIT5        0x20
#define BIT6        0x40
#define BIT7        0x80

/* END_file_bits.h */
```

In our program source file we write

```
#include "bits.h"
……..
unsigned char switches;

     switches = 0;
     switches = switches | BIT3 | BIT2 | BIT1 | BIT0;
```

Now the rightmost 4 bits are set (to 1).

A *parity bit* is a bit that is added to ensure that the number of bits with the value 1 in a set of bits is even or odd. Our set of bits shall be a byte or unsigned char. Parity bits are used as the simplest form of error detecting code.

> *even parity:* the parity bit is set to 1 if the number of ones in a byte (not including the parity bit) is odd, making the entire set of bits (including the parity bit) even.

> *odd parity:* the parity bit is set to 1 if the number of ones in a given set of bits (not including the parity bit) is even or zero, making the entire set of bits (including the parity bit) odd.

In other words, an even parity bit will be set to "1" if the number of 1's + 1 is even, and an odd parity bit will be set to "1" if the number of 1's +1 is odd.

E.2-13 We use E.2-11 to write functions computing the parity bit:

```
int even_parity(unsigned char c)
/*************************************/
/*                                   */
/* computes even parity for c        */
/*                                   */
/*************************************/
{
int  no_one_s;

     no_one_s = bitcount(c);
     if ((no_one_s % 2) != 0) {
          return 1;
     }
     else {
          return 0;
     }
} /* END_even_parity() */

int odd_parity(unsigned char c)
/*************************************/
/*                                   */
/* computes odd parity for c         */
/*                                   */
/*************************************/
{
int  no_one_s;

     no_one_s = bitcount(c);
     if ((no_one_s % 2) == 0) {
          return 1;
     }
     else {
          return 0;
     }
} /* END_odd_parity() */
```

E.2-14

There are 3 memory areas in a personal computer (PC): the main memory, the hard disk and the registers of interfaces.

The RS 232 interface is a low cost interface with a low data rate. We can use it e.g. for data exchange between a PC and a sensor/actor device.
The registers (each 1 byte length) have the following structure:

Serial Interface COM 1

03F8 RBR / THR Receive Buffer Register / Transmit Holding Register

bit 7 bit 0

03F9 IER Interrupt Enable Register

bit 7 bit 0

0	0	0	0	MS	RLS	THRE	RDA

03FA IIR Interrupt Identification Register

bit 7 bit 0

0	0	0	0	0	Interrupt ID 11->RLS 01->THRE 10->RDA 00->MS	0 means Interrupt pending

03FB LCR Line Control Register

bit 7 bit 0

Baudrate Divisor Latch	Set Break	Stick Parity	Parity 0->odd 1->even	Parity Enable 0->off 1->on	Stop Bits 0->1 1->2	Word Length 00->5 01->6 10->7 11->8

03FC MCR Modem Control Register

bit 7 bit 0

0	0	0	Loop Test	Interrupt Enable	User Defined	RTS	DTR

03FD LSR Line Status Register

bit 7							bit 0
0	Transmit Shift Register Empty	Transmit Holding Register Empty	Received Brak	Framing Error	Parity Error	Overrun Error	Data Ready

03FE MSR Modem Status Register

bit 7							bit 0
DCD	RI	DSR	CTS	Change in DCD	Change in RI	Change in DSR	Change in CTS

We will concentrate only on 3 aspects:

1. the numbers 03F8, 03F9, 03FA, …. on the left are the hex addresses of the registers on the system bus.
2. the Receive Buffer Register / Transmit Holding Register holds the byte which has been received from a connected device or which is to send to a connected device.
3. for the transmission between this PC and another device, the Line Control Register LCR defines whether each received or sent byte has a parity bit added. If a parity bit is added, the LCR defines whether this is an even or odd parity bit. Both partners on the transmission cable have to set the same parity option.

Now assume, we have read the value of the LCR into a variable unsigned char lcr_reg and we have from E.2-12:

```
#include "bits.h"
```

To adjust to no parity, we write

```
lcr_reg = lcr_reg & ~BIT3;
```

To adjust to even parity, we write

```
lcr_reg = lcr_reg |BIT4|BIT3;
```

To adjust to odd parity, we write

```
lcr_reg = (lcr_reg | BIT3) & ~BIT4;
```

Finally we have to write back the new value of lcr_reg to the LCR (which is not our topic today).

---

3 Control Flow

3.1 Switch

The switch statement is a multi-way decision that tests whether an expression matches one of a number of constant integers, and branches accordingly.

```
switch (expression) {
     case const-expr : statements
     case const-expr : statements
     default         : statements
}
```

The default branch is an option.

E.3-1

```
void   DigitAsWord(int     digit)
/************************************************/
/*                                              */
/* The value of digit is printed as word.       */
/*                                              */
/************************************************/
{
     switch(digit) {
          case 0    : printf("zero");
                      break;
          case 1    : printf("one");
                      break;
          case 2    : printf("two");
                      break;
          case 3    : printf("three");
                      break;
          ..................
          ..................
          case 9    : printf("nine");
                      break;
          default   : printf("no digit");
                      break;
     }
} /* END_ DigitAsWord() */
```

The break statement causes an immediate exit from the switch. Otherwise the execution falls through to the next statement.

Several const-expr can be grouped:

E.3-2

```
char character;

     switch(character) {
          case '{': case '}':
                    printf("curly bracket");
                    break;
          case '[':
          case ']': printf("square bracket");
                    break;
     }
```

3.2 Do-While-Loop

The do-while-loop has the termination condition at the bottom of the loop.

```
     do {
          statements
     } while (expression);
```

The statements are run at least once. The next example is itoa(), which converts a number to a char array. The digits are produced in wrong order and we have to reverse the chain of digits.

E.3-3

```
#include <string.h>
void itoa(int n, char s[])
/************************************************/
/*                                              */
/* convert n to digits s [].                    */
/*                                              */
/************************************************/
{
int i, sign;

     sign = n;
     if (sign < 0) {
          n = -n;        /* turn n positiv */
     }
     i = 0;
     do {      /* produce digits from right */
          s[i++] = n % 10 + '0'; /* next digit */
     } while ((n = n / 10) > 0);

     if (sign < 0) {
          s[i++] = '-';
     }
     s[i] = '\0';
     reverse(s);
} /* END_itoa() */
```

```
void reverse(char s[])
/***********************************************/
/*                                             */
/* Reverse array s[].                          */
/*                                             */
/***********************************************/
{
char c;
int  i, j;

     i = 0;
     j = strlen(s) - 1;
     while (i < j) {
          c = s[i];
          s[i] = s[j];
          s[j] = c;

          i++;
          j--;
     }
} /* END_reverse() */
```

We will use this example to demonstrate how we can distribute a source program to several files. Copy itoa() into a file named E_3_4a.c and copy reverse() into a file named E_3_4b.c. To use reverse() in E_3_4a.c we have to declare reverse() as extern in E_3_4a.c:

```
extern void reverse(char s[]);
```

Both files have to be translated and linked together. Using a Microsoft Visual C++ project this is done automatically by calling the compiler. Using the GNU C compiler in Linux you may translate like

```
>gcc -c E_3_4a.c
>gcc -c E_3_4b.c
```

The option -c implies generating an object. Finally you have to link

```
>gcc –o E_3_4 E_3_4a.o E_3_4b.o
```

The option -o implies that the name of the generated program will be E_3_4 instead of the standard name a.out.

E_3_4a.c could look like

E.3-4

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

extern void reverse(char s[]);

void itoa(int n, char s[])
/************************************************/
/*                                              */
/* convert n to digits s [].                    */
/*                                              */
/************************************************/
{
int i, sign;

     sign = n;
     if (sign < 0) {
          n = -n;        /* turn n positiv */
     }

     i = 0;
     do {      /* produce digits from right */
          s[i++] = n % 10 + '0'; /* next digit */
     } while ((n = n / 10) > 0);

     if (sign < 0) {
          s[i++] = '-';
     }
     s[i] = '\0';

     reverse(s);
} /* END_itoa() */

int main(void)
/***************************************/
/*                                     */
/* Test of distribution of source code  */
/*                                     */
/***************************************/
{
char  digits[10];

     /* First initialize digits with 0 */
     /* memset() needs <string.h>      */
     memset(digits, 0, sizeof(digits));
     itoa(245, digits);
     printf("Number as characters: %s\n", digits);
     return(0); /* to satisfy compiler */
} /* END_main() */
```

3.3 Break, Continue and Exit

The break statement commands an exit from for-, while- and do-loops as well as from inside a switch statement. From the point of the break a jump after the end of the enclosing loop or switch is performed.

The continue statement inside a loop causes the immediate start of the next iteration.

.E.3-5 Stepping through an array int value[].
Negative values are ignored.

```
for (i = 0; i < n; i++) {
     if (value[i] < 0) {
          continue;
     }
     ..... non negative values are processed .....
}
```

With `exit(0);`

a program can be finished in every line of code.

To use exit(), you have to

#include <stdlib.h>

3.4 Goto and Labels

'C' has like many other languages the dangerous goto statement. This causes a jump to a label, which is a name followed by a colon, marking a statement.

There is only one reason to use a goto in 'C'. That is to leave nested loops.

E.3-6

```
     for ( ..... ) {
          for ( ......... ) {
               ............
               goto loops_end;
               ............
          }
     }
loops_end: next statements;
```

---

4 Program Structure

4.1 Recursion

'C'-functions can be called recursively, that is, a function can call itself. This can happen directly or via a second function indirectly. One of the most famous examples for recursive calls is the computation of n! = n * (n-1) * (n-2) * ... * 1.

E.4-1

```
unsigned int  faculty(unsigned int n)
/*************************************************/
/*                                               */
/* Computation of n! with recursive function calls. */
/*                                               */
/*************************************************/
{
     if((n == 1) || (n == 0)){
          return 1;
     }
     else {
          return(n * faculty(n-1));
     }
} /* END_faculty() */
```

Make a trace of the algorithm with paper and pencil for the computation of 4! to understand what happens.

The next example for recursion is the alternative implementation of E.3-4: printing a number character by character on to the monitor. By integer division we get the last character first, but we want to print the most significant character first.

E.4-2

```
#include <stdio.h>

void PrintDigit(int n)
/*************************************/
/*                                   */
/* prints n character by character.  */
/*                                   */
/*************************************/
{
     if (n < 0) {
          putchar('-');
          n = -n;
     }
     if (n/10 != 0) {
          PrintDigit(n/10);
     }
     putchar(n%10 + '0');
} /* END_PrintDigit() */
```

Make a trace of the algorithm with paper and pencil for PrintDigit(243) to understand what happens.

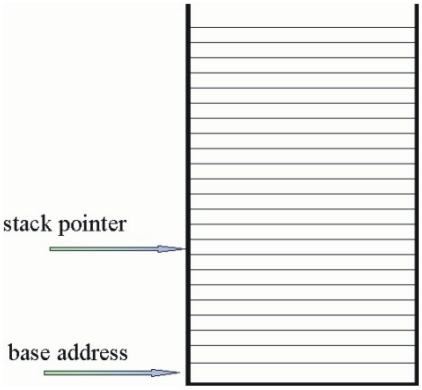
Another example about recursion will follow in the chapter about pointers.

4.2 Stack

4. 2 Stack

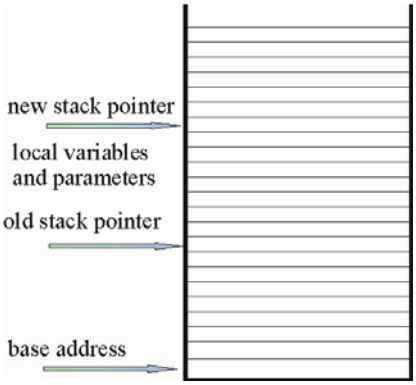
When a program starts, it has two memory areas in the main memory: the heap and the stack. A global variable is placed in the heap and lives as long as the program lives.

A stack is a contiguous piece of memory starting with a base address. The stack has the ability to increase and to decrease. The stack pointer shows the address up to which the stack is used.



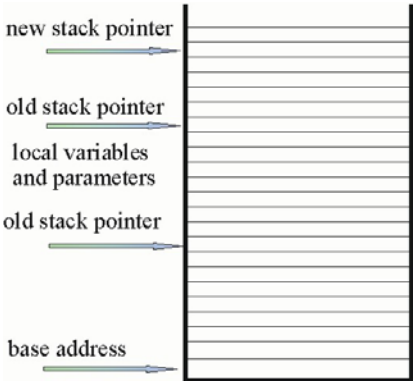
Pic.4-1a

When the program execution enters a function, the stack increases and the local/automatic variables are placed on the stack as well as copies of parameters out of the parameter list.



Pic.4-1b

Calling another function from inside this function lets the stack increase again.



Pic.4-1c

Leaving the last function decreases the stack to Pic.4-1b and leaving the first function again decreases the stack to Pic.4-1a. The freed stack bytes are not set to zero, that means, entering another function and forgetting to initialise the local variables lets the local variables take the old values which just happen to be on the stack.

Now try to imagine what happens on the stack in E.4-1.

Be careful using recursion in 24 h x 7 days applications. The amount of memory is limited and stack overflow causes a program crash. Be sure that your recursions end after a finite number of steps or avoid recursion.

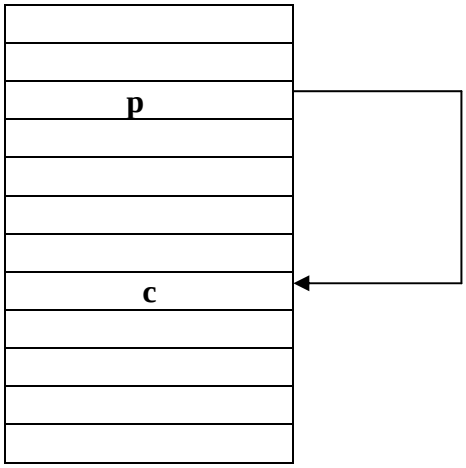
5 Pointer

5.1 Pointer and Addresses, Pointer Arithmetic

A pointer is a variable that contains the address of an object.

The unary address operator '&' assigns the address of c to the variable p. p is said to point to c.

main memory:



The unary operator '*' is the indirection or dereferencing operator. Applied to a pointer p gives access to the variable to which p points.

E.5-1 some code line examples

```
int  x, y;
int  z[10];
int  *ipoint;  /*  ipoint is a pointer to int  */

     x = 1;
     y = 2;
     ipoint  = &x;      /* ipoint points to x    */
     y       = *ipoint; /* y is 1                */
     *ipoint = 0;       /* x is 0                */
     ipoint  = &z[0];   /* ipoint points to z[0] */
```

ipoint can only point to int objects. A pointer to float objects has to be defined as

```
float     *fpoint;
```

The last line of E.5-1 can also be written as

```
ipoint = z;
```

The name of an array serves also as a pointer to the first element of that array.

Pointer operations with the definitions of E.5-1:

```
ipoint = z;
```

```
y       = *ipoint + 1;
```

The value 1 is added to z[0], the result is stored in y.

```
(*ipoint)++;
```

increases the value of z[0] by 1. Attention:

```
y = *(ipoint++) + 1;
```

adds 1 to z[0] and this result is stored in y. Afterwards ipoint points to z[1];

In general the operations pointer++ depends on the type of the object on which pointer points.
If pointer points to a variable of type char, pointer++ increases the address by 1.
If pointer points to a variable of type short, pointer++ increases the address by 2.
If pointer points to a variable of type float, pointer++ increases the address by 4.

E.5-2

A message (a string of bytes) comes from a network into our computer and is stored in an array telegram[]. To process this message, we let a pointer point to the first component of this message.

```
unsigned char telegram[200];
unsigned char *pointer;

     pointer = telegram;
```

Example of the byte wise structure of a message in an industrial communication network:

pointer	standardized message type ID
pointer++	length of payload information
pointer++	segment number in case of several segments
pointer++	message number
pointer++	sender ID
pointer=pointer+2	receiver ID
pointer=pointer+2	sensor state, e.g. normal or failure
pointer=pointer+1	date
pointer=pointer+3	time
pointer=pointer+3	data payload. the amount of the following bytes is given by byte 2 of this message. Contents e.g. sensor data.

The value of pointer has been changed in every step.

E.5-3

Let there be an int array numbers[] and an int pointer, pointing to numbers[0]. We assume the size of an int as 4 bytes. Then we have the following address structure in memory:

```
int  numbers[10];
int  *ipointer;

     ipointer = numbers;
```

ipointer points to	numbers[0]	startaddress
ipointer+1 points to	numbers [1]	startaddress + 4
ipointer+2 points to	numbers [2]	startaddress + 8
ipointer+3 points to	numbers [3]	startaddress + 12

The value of ipointer has not changed.

Another example to work with pointers is

```
int  *ipoint1, *ipoint2;

     ..........................................................
     ipoint2 = ipoint1;
```

ipoint2 points now to the same object as ipoint1 dose.

Note with the above definitions: redirecting numbers like

```
     numbers = ipointer + 4;       /* error */
```

is not possible. The name of an array has not all properties of a pointer.

Char pointer can point to string constants, which are arrays of characters like

```
     "This example"
```

In the internal representation, the array is terminated with '\0'. Example:

```
char *pmessage;

     pmessage = "first message";
     printf("%s\n",pmessage);
```

A pointer initialisation with a string constant can be done in the definition header of a function:

```
char message[] = "second message"; /* an array  */
char *pmessage = "third message";  /* a pointer */

     printf("%s\n%s\n",message, pmessage);
```

The compiler calculates the length of message[] so that all characters including the final '\0' fit into the array.

5.2 Pointers and Function Arguments

In case of a function call 'C' copies the parameter values onto the stack and the function algorithm is performed on the stack variables. This is a call by value and the called function cannot alter the variables of the calling function.

In order to give back value changes in variables to the outside of the called function, we have to use pointer in the parameter list. This is a call by reference.

E.5-4

```
void swap(int *px,
          int *py )
/************************************************/
/*                                              */
/* swapping the values of two int variables.    */
/*                                              */
/************************************************/
{
int  temp;

     temp = *px;
     *px  = *py;
     *py  = temp;

} /* END_swap() */
```

Example for the call:

```
int  ia, ib;

     ia = 6;
     ib = 8;
     swap(&ia, &ib);
```

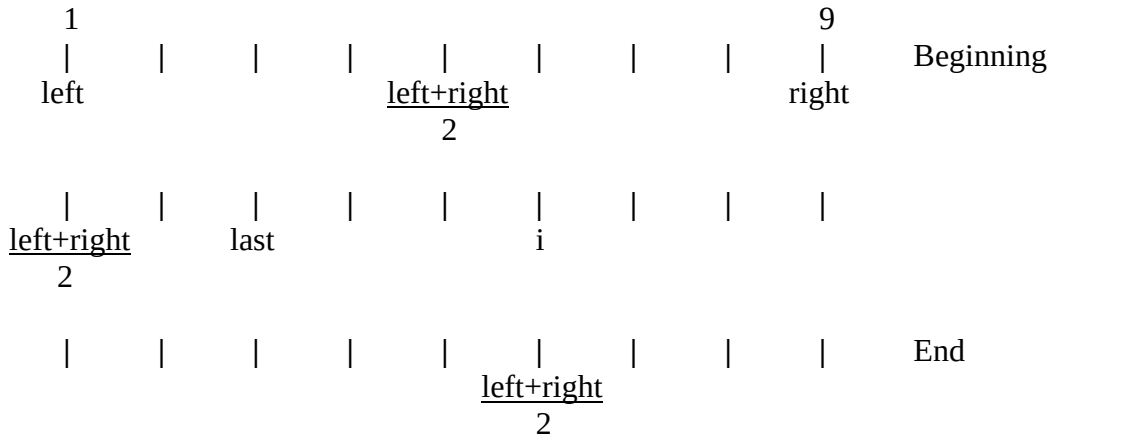
Because an array name acts also as a pointer to the first component of the array, an array is passed to a function as call by reference and the function algorithm processes directly on the components of the array. Arrays as parameters are not copied on to the stack.

E.5-5 Quicksort

An int array shall be sorted into increasing order. From the array, the element in the middle is chosen and the other array elements are partitioned into two subsets. One subset contains all elements which are less then the chosen element, the other subset contains the elements which are greater than or equal to the chosen element. The same procedure is recursively applied to both subsets. Continuing in this way we will eventually have subsets with fewer than two elements. This ends the recursion.

```
void swap(int *px, int *py );   /* from E.5-4 */


void qsort(int v[], int left, int right)
/*****************************************************/
/*                                                   */
/* Sort v[left],...,v[right] in increasing order.    */
/*                                                   */
/*****************************************************/
{
int  i, last;

     if(left >= right) {
          return;    /* nothing to sort */
     }
     swap(&v[left], &v[(left+right)/2]);
     last = left;
     for(i=left+1; i <= right; i++){
          if(v[i] < v[left]){
               swap(&v[++last], &v[i]);
          }
     }
     swap(&v[left], &v[last]);

     qsort(v, left, last-1);
     qsort(v, last+1, right);

} /* END_qsort() */
```

Steps in the sorting algorithm:



5.3 Formatted Input with scanf()

The input function *scanf()* is analogous to *printf()*.

```
int  scanf(char *format, &arg1, &arg2, ...);
```

scanf() reads characters or character strings from the keyboard and converts it according to information given by format. Characters and character strings are separated by blank, tab or '\n'.

The result of the function is the number of successfully scanned input fields.
sscanf() acts in a similar way but reads not from the keyboard but instead out of a character string:

```
int  sscanf(char *string, char *format, &arg1, &arg2, ...);
```

The format string can contain:

- blanks or tabs which are ignored
- characters which correspond to the next characters in the input stream
- format definitions starting with % in analogy to *printf()*

E.5-6

a)
```
int     i;
float   f;

    scanf("%d  %f", &i, &f);
    printf("i = %d, f = %f", i, f);
```

input e.g. 3 4.7

b)
```
int     i;
float   f;

    scanf("%d and %f", &i, &f);
    printf("i = %d, f = %f", i, f);
```

input e.g. 3 and 4.7

E.5-7

a) Expected is

25 th Dec 2010

The *scanf()*-call is:

```
int  day, year;
char monthname[20];

     scanf("%d th %s %d", &day, monthname, &year);
```

b) Expected is the american date format mm/dd/yy

The *scanf()*-call is:

```
int  day, month, year;

     scanf("%d/%d/%d", &month, &day, &year);
```

A problem is now the input of a line of chars including an unknown number of blanks. We need a special conversion specification or we use another function.
The conversion specification is: `%[^\n]`

E.5-8

```
#include <stdio.h>

int main(void)
/*****************************************************/
/*                                                   */
/* 3 ways to read a line of text.                    */
/*                                                   */
/*****************************************************/
{
char   text1[80];
char   text2[80];
char   text3[80];
int    i;

     printf("enter line 1 of text\n");
     scanf("%[^\n]", text1); getchar();  /* !! */
     printf("\ntext1 = %s\n", text1);

     printf("enter line 2 of text\n");
     fgets(text2, sizeof(text2), stdin);
     printf("\ntext2 = %s\n", text2);

     printf("enter line 3 of text\n");
     i = 0;
     text3[i] = getchar();
     while (text3[i] != '\n') {
          i++;
          text3[i] = getchar();
     }
     text3[i] = '\0';
     printf("\ntext3 = %s\n", text3);

     return  0; /* to satisfy compiler */
} /* END_main() */
```

Don't forget getchar() after scanf(), otherwise '\n' is still in the input buffer.

5.4 Command-line Arguments

Typing a command in the system window offers the possibility to pass parameters to the command. These are treated as parameters to main() and are named argc and argv. argc is the number of strings in the command line including the name of the program. argv is a pointer to an array of pointers to char strings containing the program name and the parameters. argv[0] points to the name of the command (program).

E.5-9

a)

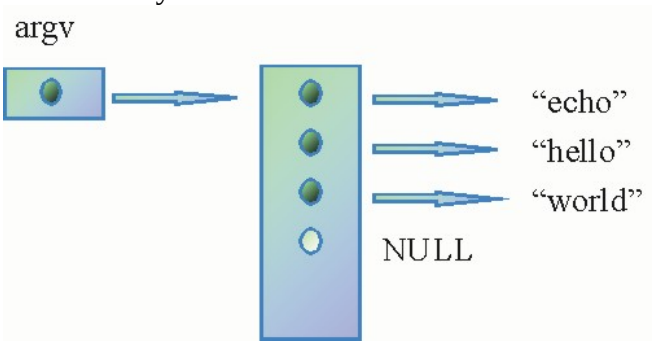
We write a program *echo*, which writes the subsequent parameters on to the monitor. The command

```
>echo hello world
```

produces the output

```
hello world
```

argc is 3 and argv is illustrated by



Pic.5-1

argv[argc] is the NULL pointer.

```
#include <stdio.h>

int  main(int  argc,
          char *argv[])
/************************************/
/*                                  */
/* Echo of command line parameters. */
/*                                  */
/************************************/
{
int  i;

     for (i = 1; i < argc; i++) {
          printf("%s  ",argv[i]);
     }
     printf("\n");
     return  0; /* to satisfy compiler */
} /* END_main() */
```

b)

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

int main(int argc, char *argv[])
/************************************/
/*                                  */
/* 2 numbers as parameters.         */
/* print the sum of them.           */
/************************************/
{
int first, second;
     if (argc < 3){
          printf("parameters missing");
          exit(-1);
     }
     first = atoi(argv[1]);
     second = atoi(argv[2]);
     printf("\nsum = %d \n", first + second);
     if (argc > 3){
          printf("other parameters ignored");
     }
     return  0; /* to satisfy compiler */
} /* END_main() */
```

5.5 Pointer to Functions

We can define pointers to functions which are passed as parameters to other functions. In the parameter list the formal parameter is defined as pointer to function, the actual parameter is just the name of a function.

A function in a parameter list is a so-called 'callback function'.

E.5-10 Print a line with some function values for several functions.

```
#include <stdio.h>
#include <math.h>

double square(double x)
/************************************/
/*                                  */
/* f(x) = x * x                     */
/*                                  */
/************************************/
{
     return(x * x);
} /* END_square() */

void values(double (*f)(double x) )
/************************************/
/*                                  */
/* print f(x), x = 0.1 to 1.0       */
/*                                  */
/************************************/
{
int    i;
double y, h;

     for (i = 1; i <= 10; i++) {
          h = i/10.0;
          y = (*f)(h);
          printf("%5.2f, ", y);
     }
     printf("\n");
} /* END_values() */
```

```
int main()
/************************************/
/*                                  */
/* values of x*x, sin(x), exp(x)    */
/*                                  */
/************************************/
{
     values(square);
     values(sin);   /* sine-function from library */
     values(exp);.. /* exp-function from library */
..   return  0; /* to satisfy compiler */
} /* END_main() */
```

Inside the function the pointer to function is used with the dereferencing operator '*' like other call by reference parameters.

6 Multi-dimensional Arrays

First we are coming back to arrays of char pointers which result in a 2-dimensional object.

```
char    *error[3]
```

error is an array with 3 pointers. These can point e.g. to error messages:

```
error[0] = "temperature error";
error[1] = "voltage error";
error[2] = "connection error";

printf("%s \n", error[0]);
```

The 2. dimension is variable.

Now we will define multi-dimensional number arrays. We regard 2-dimensional arrays. A 4x4 - matrix with int components is defined as:

```
int  forxfor[4][4]; /* [row][column] */

     forxfor[2][3] = 27;
```

E.6-1
Date conversion from day of the month to day of the year and vice versa.

Remark: "&&" has higher precedence than "||".

```
char daytab[2][13] = {
     {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
     {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};

int day_of_year(int  year,
                int  month,
                int  day   )
/*********************************************/
/*                                           */
/* day of year from month and day.           */
/*                                           */
/*********************************************/
{
int  i, leap;
     if ((year%4 == 0) && (year%100 != 0) || (year%400 == 0)) {
          leap = 1;
     }
     else {
          leap = 0;
     }
     for (i = 1; i < month; i++) {
          day = day + daytab[leap][i];
     }
     return day;
} /* END_day_of_year() */
```

```
void month_day(int  year,
               int  yearday,
               int  *pmonth,
               int  *pday   )
/*********************************************/
/*                                           */
/* month and day from day of year.           */
/*                                           */
/*********************************************/
{
int  i, leap;

     if ((year%4 == 0) && (year%100 != 0) || (year%400 == 0)) {
          leap = 1;
     }
     else {
          leap = 0;
     }
     for (i = 1; yearday > daytab[leap][i]; i++) {
          yearday = yearday - daytab[leap][i];
     }
     *pmonth = i;
     *pday = yearday;

} /* END_month_day() */
```

Passing a 2-dimensional array like daytab [][] to a function, the parameter declaration in the function must include the number of columns (2. dimension), the number of rows is irrelevant. (For information: what is passed is a pointer to an array of rows, where each row is an array of 13 ints.) Passing daytab to a function f, the declaration of daytab would be

```
     f(int daytab[2][13])  { ... }
```
or
```
     f(int daytab[][13])   { ... }
```

Using more-than-2-dimensional arrays means only the first dimension is irrelevant.

Next is an example of the initialization of an array of char pointers with text strings.

E.6-2

```
char *month_name[] ={ /* the dimension is determined by the
                         compiler.                          */
     "illegal month",
     "January", "February", "March",
     "April", "May", "June",
     "July", "August", "September",
     "October", "November", "December"
};
```

---

7 Structures

7.1 Definition and Access

A structure is a set of one or more variables. They can be of different types. Each member of the structure can be accessed. The structure has a name, a structure tag. Example:

```
struct complex {
          float     re; /* real part      */
          float     im; /* imaginary part */
       };
```

The structure declaration defines a type. There is no memory space reserved. After the last brace may follow a list of variables:

```
struct complex {
          float     re;
          float     im;
       } x, y, z;
```

x, y and z are variables of type complex with memory space reserved for them. The structure tag can also be used to define variables. An equivalent definition of x, y and z is:

```
struct complex x, y, z;
```

a structure member is accessed with a dot notation: `x.re` or `x.im`

E.7-1

```
#include <stdio.h>

int main(void)
/************************************ */
/*                                    */
/* complex numbers as struct          */
/*                                    */
/************************************ */
{
float  abs_value;

struct complex {
          float     re;
          float     im;
       } x, y, z;

     x.re = 2.1;
     x.im = 1.1;
     y = x;
     printf("Output of the complex number y: \n");
     printf("real part: %6.3f, imaginary part: %6.3f\n",
             y.re, y.im                                 );

     /* 2 * x */
     x.re = x.re * 2.0;
     x.im = x.im * 2.0;
     /* z = x + y */
```

```
     z.re = x.re + y.re;
     z.im = x.im + y.im;
     printf("Output of the complex number z: \n");
     printf("real part: %6.3f, imaginary part: %6.3f\n",
             z.re, z.im                                  );

     /* absolut value of z */
     abs_value = sqrt((double)(z.re*z.re+z.im*z.im));
     printf("absolut value of z: %6.3f\n", abs_value);
     return 0;
} /* END_main() */
```

Another example is a struct for the payroll record of an employee:

E.7-2

```
struct staff {
          char           family_name[20];
          char           given_name[20];
          struct {
                    unsigned short year,
                                   month,
                                   day;
                 } birthday;
          float          salary;
       };

struct staff   employee_1, employee_2;

     employee_1.birthday.month = 12;
```

The size of a struct in no. of bytes is given in two ways:

```
printf("size with tag: %d, size of variable: %d\n",
        sizeof(staff),     sizeof(employee_1)      );
```

7.2 Structures, Functions, Pointers, Arrays

The handling as a function parameter or as a function type is in analogy to scalar data types.

E.7-3

```
struct complex c_add(struct complex    x,
                     struct complex    y)
/************************************ */
/*                                    */
/* add two complex numbers            */
/*                                    */
/************************************ */
{
struct complex temp;

     temp.re = x.re + y.re;
     temp.im = x.im + y.im;
     return temp;
} /* END_c_add() */
```

Here we assume a global declaration of struct complex.

Structure pointers are similar to pointers to scalar data types. The next example shows the definition of a pointer to structure and the access of a struct member with a pointer.

E.7-4

```
struct complex z;
struct complex *c_pointer;


     c_pointer = &z;
     printf("Output of the complex number z: \n");
     printf("real part: %6.3f, imaginary part: %6.3f\n",
             (*c_pointer).re, (*c_pointer).im         );
```

An equivalent notation for accessing members with pointers is

`c_pointer->re` and `c_pointer->im`

Arrays of structures are defined in analogy to arrays of scalar types. Assume a set of 4 complex numbers:

```
struct complex numbers[4];

     numbers[1].re = 6.5;

     size_no_of_bytes = sizeof(numbers);
```

7.3 Typedef

'C' provides the keyword `typedef` for declaring new data types. The new data types are derived from known data types and include enumeration types. The declaration

```
typedef int Length;
```

makes the name Length a synonym for int. The type Length can be used in definitions, declarations and casts:

```
Length   len, maxlen;
Length   *pointer_to_len;
```

The declaration

```
typedef char *String;

String       pointer_to_char;
```

makes String a synonym for a char pointer.

E.7-5

a)
```
typedef int  Kilometer;

Kilometer trip_begin, trip_end;
```

b)
```
typedef struct { /* same members as in  E.7-2 */
          char           family_name[20];
          char           given_name[20];
          struct {
                    unsigned short year,
                                   month,
                                   day;
                 } birthday;
          float          salary;
       } MasterData;

MasterData     employee_1, employee_2;
MasterData     extern_employee[6];
```

Access is in analogy to E.7-2.

```
extern_employee[3].birthday.month = 10;
extern_employee[3].salary = 3421.60;

printf("extern no 3 birth month: %2d\n",
        extern_employee[3].birthday.month);
printf("extern no 3 salary: %7.2f\n",
        extern_employee[3].salary    );
```

c)
```
typedef char NameType[20];

NameType  family_name, given_name;
```

Our MasterData struct reads now:

```
typedef struct { /* same members as in  E.7-2 */
          NameType       family_name;
          NameType       given_name;
          struct {
                    unsigned short year,
                                   month,
                                   day;
                } birthday;
          float          salary;
       } MasterData;

     employee_1.family_name[0] = 'N';
```

---

s

E.7-6

```
a) typedef enum {YELLOW, RED, BLUE} Colour;

b) typedef enum {FALSE, TRUE} Boolean;

Boolean check;

     check = TRUE;
```

Please note: a typedef declaration does not create a new type in any sense, it derives new names from existing types. This provides better documentation for a program. typedef is different to #define. typedef is interpreted by the compiler, #define is interpreted by the preprocessor.

7.4 Memory Alignment

Memory for variables is often grouped inside a computer. Assume the definition of a float variable. The variable is stored in a new group of bytes. It is possible that the last group contains bytes which are not allocated for a variable and thus stay free. Several char variables can be contained in one group.
If available memory is a concern for you, you should have in mind the next example.

E.7-7

```
struct sensor_value_1 {
          unsigned char address;
          float         value;
          unsigned char state;
} sv_1;
```

`sizeof(sv_1)` indicates, that 12 bytes are needed.

```
struct sensor_value_2 {
          unsigned char address;
          unsigned char state;
          float         value;
} sv_2;
```

`sizeof(sv_2)` indicates, that 8 bytes are needed!

We can even add two chars without enlarging the memory for the struct:

```
struct sensor_value_3 {
          unsigned char address;
          unsigned char state;
          unsigned char hour;
          unsigned char minute;
          float         value;
} sv_3;
```

`sizeof(sv_3)` indicates, that again 8 bytes are needed.

7.5 Unions

A union is a variable that may hold at different times objects of different types and size.

The compiler keeps track of size and alignment . Using a union may save memory.

Assume, we have a communication protocol between two computers in a network and for some reason the protocol has to have a constant length. The payload might be either of type int or of type float or of type char.

```
union u_tag {
     int       ival;
     float     fval;
     char      cval;
} payload;
```

The compiler stores enough memory to hold the largest of the 3 types. Any one of these types may be assigned to payload and used in expressions, so long as the usage is consistent.

It is the programmer's responsibility to keep track of which type is currently stored in a union. If something is stored as one type and extracted as another type, the result is garbage.

Members of a union are accessed as like as members of structures:

```
     union_name.member
```

or

```
     union_pointer->member
```

If the variable utype is used to keep track of the type in payload, a code example cold be:

```
enum {INT, FLOAT, CHAR} utype;

     if (utype == INT) {
          printf("%d\n", payload.ival);
     }
     else if (utype == FLOAT) {
          printf("%f\n", payload.fval);
     }
```

```
     else if (utype == CHAR) {
          printf("%c\n", payload.cval);
     }
     else {
          printf("unknown type in union.\n");
     }
```

`sizeof payload` gives 4 bytes amount of memory for payload.

The same operations are permitted on unions as on structures: assignment of unions, taking the address and accessing a member.

E.7-8

The following example is similar to E.5-2. It might appear in a simplified building automation communication protocol. A floor substation notices a change in room temperature and sends an event to the building control central computer. This message has constant length independent of the type of information being sent.

```
#define   DIGITAL_INPUT  0
#define   ANALOG_INPUT   1
#define   TEXT           2


typedef struct BA_event {
          unsigned char  standard_messageID;
          unsigned char  message_type; /* see above defines */
          unsigned char  segment_number;
          unsigned char  message_number;
          unsigned short senderID;
          unsigned short receiverID;
          unsigned char  sensor_state; /* normal or failure */
          struct {
               unsigned char  day;
               unsigned char  month;
               unsigned char  year;
          } date;
          struct {
               unsigned char  hour;
               unsigned char  minute;
               unsigned char  second;
          } time;
          union {
               short     digital_value;
               float     analog_value;
               char      message_text[20];
          } payload;
} BA_event;
```

`sizeof BA_event` gives 36 bytes for storage.

7.6 Bit-fields

It may be useful to pack a set of single-bit flags into 1 variable. An example is the contents of a hardware register as in E.2-14. Another example is the information about a technical device,

e.g. a sensor. We can encode the information in one char and then we can operate on the char with a set of masks corresponding to the bit positions as in

```
#define OPERATIONAL      0x01
#define AUTOMATIC        0x02
#define ON               0x04
```

Then accessing the bits becomes a matter of masking. We may have

```
unsigned char device;

     device = device | AUTOMATIC | ON;
```

turning on the AUTOMATIC and ON bits in device.

```
     device = device & ~(AUTOMATIC | ON);
```

turns the bits off, and

```
     if ((device & (AUTOMATIC | ON)) == 0) { …
```

is true if both bits are off.

As an alternative C offers the capability of accessing bit-fields within a memory word directly. A bit-field is a set of adjacent bits within a single implementation defined storage unit (a word). The syntax of a field is based on structures. The code is now:

```
struct {
     unsigned int operational : 1; /* operational (1) or
                                      failure (0)        */
     unsigned int automatic   : 1; /* automatic (1) or
                                      manual (0)         */
     unsigned int on          : 1; /* on (1) or off (0)  */
} sensor;
```

sensor is a variable containing three 1-bit fields. The number after the colon is the field width. Although defined as unsigned int, each member has only one bit.

Individual fields are referenced as structure members:

```
     sensor.operational   or   sensor.automatic
```

Fields behave like small integers and may be included in arithmetic expressions. Instead of bit masking we may write now

```
     sensor.automatic = 1;
     sensor.on        = 1;
```

to turn bits on, or

```
     sensor.automatic = 0;
     sensor.on        = 0;
```

to turn bits off, and

```
     if (sensor.automatic == 0 && sensor.on == 0) { …
```

to test bits.

`printf("size sensor: %d\n", sizeof sensor);` gives 4 bytes.

```
memset(&sensor, 0, sizeof(sensor));
sensor.operational = 1;
sensor.automatic = 0;
sensor.on = 1;
```
`printf("sensor %x\n", sensor);` gives 5 on my computer.

```
int mult = 3;
```
`printf("sensor.on * mult %d\n", sensor.on * mult);` gives 3

```
/* sensor = 0; is not allowed*/
```

Bit fields improve readability, but be careful, almost everything about fields is implementation dependent. Before you exchange bit-fields between computers you have to find out, how the order of bytes is arranged in a word. ´

Arrays of bit fields is implementation dependent. The use of the address operator '&' is implementation dependent. (On my computer it works)

Finally we assume that our sensor can detect 4 different states, named as 0, 1, 2, 3. We extend our bit-field structure:

```
struct {
     unsigned int operational : 1; /* operational (1) or
                                      failure (0)        */
     unsigned int automatic   : 1; /* automatic (1) or
                                      manual (0)         */
     unsigned int on          : 1; /* on (1) or off (0)  */
     unsigned int state       : 2; /* states are 0,1,2,3 */
} sensor2;

     sensor2.state = 3;
```

2 bits are reserved for state.

7.7 Linked List

With pointers and with the functions

```
     void *malloc(size_t  size);

     void free(void *buffer);
```

we can allocate and free memory space dynamically. We need an `#include <stdlib.h>`

The actual parameter for *malloc()* is just a positive int. *mallco()* reserves at least *size* bytes. They are not initialised. These bytes can be accessed via a pointer. Internally more bytes can be reserved because of word boundaries.

The memory is allocated in the heap and not in the stack. Therefore the memory remains allocated after leaving the function, where *malloc()* has been called. *free()* deallocates the memory.

malloc() returns a void pointer which will be casted to the appropriate type.

E.7-9

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

int *pi;
          pi = (int *)malloc(sizeof(int));
          *pi = 45;
          printf("*pi = %d\n", *pi);
          free(pi);
```

Don't access the freed memory block with pi any more!

The next subject is a structure, where one of its members is a recursive pointer to just this structure type

```
struct example {
     int  dummy;
     struct example *rec_point;
}
```

rec_point is a pointer to a structure, not a structure itself. Also an implicit recursive pointer definition is possible like:

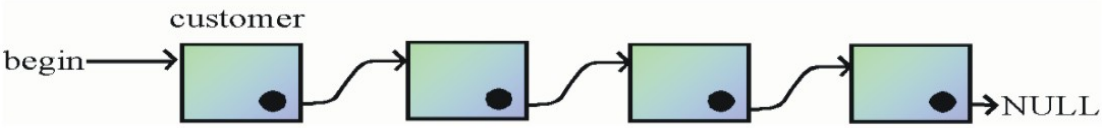
```
struct t {
     ..................
     struct s *p;
};
struct s {
     ..................
     struct t *q;
};
```

A linked list is a sequence of structures each containing a pointer, pointing to the next structure.

E.7-10 We will handle a list with customer names.

```
#define NAMELENGTH   27
typedef char            namearray[NAMELENGTH];
typedef struct customer *customerpointer; /* this is legal */
typedef struct customer{
                  namearray       name;
```

```
                    customerpointer next;
                } customer;

customer  *begin;
```



Pic.7-1a

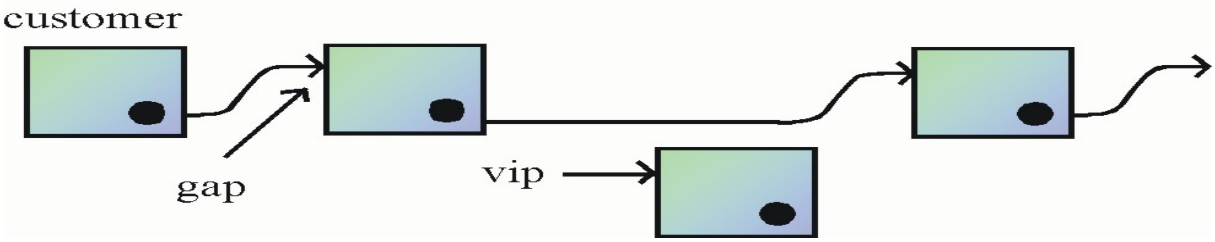
begin points to the 1. customer,
begin->next points to the 2. customer,
begin->next->next points to the 3. customer.

Insert a new customer at the point 'gap'

Assume

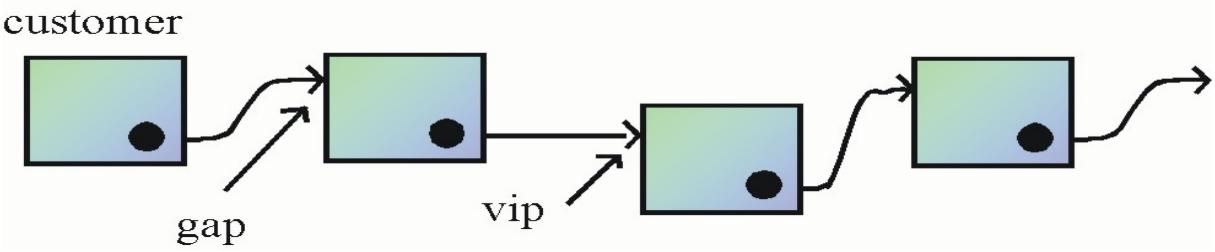
```
customer  *gap, *vip;
```

gap points to the customer after whome the new customer shall be inserted, vip points to the new customer.



Pic.7-1b

```
vip = (customer *)malloc(sizeof(customer));
vip->name to be read in;
vip->next = gap->next;
gap->next = vip;
```

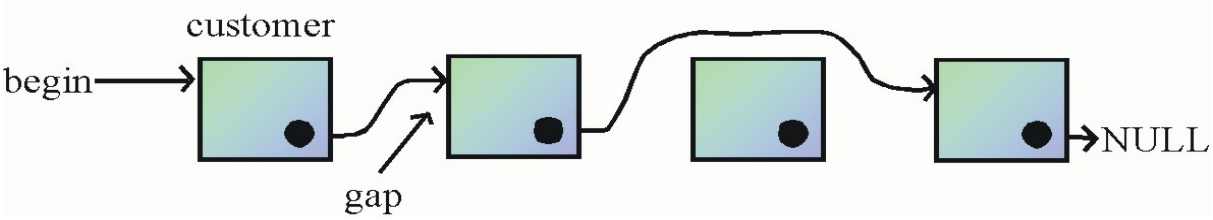


Pic.7-1c

.Delete a customer

a) the bad variant:

```
gap->next = gap->next->next;
```



Pic.7-1d

Now there is no link to the removed customer. The memory is still reserved and can not be freed, it is no more available.

b) the better variant:

```
customer *nobody;
nobody = gap->next;
gap->next = gap->next->next;
free(nobody);
```

For the runtime system remains the problem of the garbage collection: after several *malloc()* and *free()* with memory blocks of different size, there are holes in the heap. There may be not enough contiguous memory for the next *malloc()*, although the sum of the holes would be large enough.

E.7-11 A program with a linked list

```
/*****************************************************************************/
/*                                                                           */
/*      file : E_7_11.c                                                      */
/*      Version 2.0                                                          */
/*                                                                           */
/*      (c) 2010, B.Guesmann                                                 */
/*      Date   :   08.04.2010                                                */
/*                                                                           */
/*      Customer names in a linked list.                                     */
/*      Append a new customer and delete an old customer.                    */
/*                                                                           */
/*****************************************************************************/
#include <stdio.h>
#include <stdlib.h>

/*************** File E_7_11.c contains ******************/
void readname       (void);
void append_customer(void);
void delete_customer(void);
int  main           (void);

/********************** defines ***************************/
#define NAMELENGTH     27

/********************** typedefs **************************/
typedef char            namearray[NAMELENGTH];
typedef unsigned int    natural;
typedef struct customer *customerpointer; /* this is legal */
typedef struct customer{
                  namearray       name;
                  customerpointer next;
               } customer;

/************ external (global) Variables  ****************/
customer        *begin, *end;
namearray       name_input;

/********************** Functions *************************/

void readname(void)
/************************************************/
/*                                              */
```

```
/* read a name from keyboard                          */
/*                                                    */
/******************************************************/
{
natural i;
char    c;

        for (i = 0; i < NAMELENGTH; i++) {
                if ((c = getchar()) == '\n') {
                        name_input[i] = '\0';
                        break;
                }
                else {  name_input[i] = c;
                }
        }
        name_input[NAMELENGTH-1] = '\0';
} /* END_readname() */

void append_customer(void)
/******************************************************/
/*                                                    */
/* Append new customer at the end of the list.        */
/*                                                    */
/******************************************************/
{
customer  *new_customer;
natural   n;

        new_customer = (customer *) malloc(sizeof(customer));
        if (begin == NULL) {
                begin = new_customer;
        }
        else {  end->next = new_customer;
        }
        for (n = 0; n < NAMELENGTH; n++) {
                new_customer->name[n] = name_input[n];
        }
        new_customer->next = NULL;
        end = new_customer;
} /* END_append_customer() */

void delete_customer(void)
/******************************************************/
/*                                                    */
/* Delete old customer at the beginning of the list*/
/*                                                    */
/******************************************************/
{
customer   *served_customer;

        if (begin != NULL) {
                served_customer = begin;
                begin = begin->next;
                printf("served customer was : %s\n",
                                served_customer->name);
                free(served_customer);
        }
        else {  printf("no customer available!\n");
        }
} /* END_delete_customer() */
```

```
int main(void)
/*****************************************************/
/*                                                   */
/* Append and delete customer in linked list         */
/*                                                   */
/*****************************************************/
{
char    readnext;

        begin = NULL;
        readnext = 'y';
        while (readnext == 'y' ) {
                printf("please enter name:");
                readname();
                append_customer();
                printf("next customer ? y or n :");
                readnext = getchar();
                getchar(); /* for enter key */
        } /* end while */
        printf("finishing time? y or n :");
        readnext = getchar();
        getchar(); /* for enter key */
        while (readnext == 'n') {
                printf("customer is served and deleted.\n");
                delete_customer();
                printf("finishing time? y or n :");
                readnext = getchar();
                getchar(); /* for enter key */
        }
        printf("well done  ");
        return 0;
} /* END_main() */
/* END_file_E_7_11.c */
```

The linked list in E.7-11 permits a forward search through the list from begin to end. For a backward search we would need a 2. pointer to build a backward link.

A final remark: I don't recommend *malloc()* and *free()* in 24 h x 7 d - applications.

8 Input and Output

8.1 Standard I/O

The library implements a simple model of text input and output. A text stream consists of a sequence of lines, each line ends with a '\n'. The library might convert <carriage return> and <linefeed> to newline on input and back again on output.

```
int getchar(void);
```

reads one char from standard input. Redirecting the input from the keyboard to a file looks like

```
program < infile
```

Now *getchar()* in *program* reads from *infile* and not from the keyboard. The last object read in is EOF: End Of File, which is defined in <stdio.h>. EOF is more than 1 byte. That's why there should be an int on the left side of *getchar()*.

putchar(c); puts c on the standard output.

E.8-1

```
#include <stdio.h>
#include <ctype.h>

int main(void)
/*************************************/
/*                                   */
/* convert input to lower case output */
/*                                   */
/*************************************/
{
int   c;

     while ((c = getchar()) != EOF) {
          putchar( tolower(c) );
     }
     return 0;
} /* END_main() */
```

To generate EOF from the keyboard type <Strg><z> .

8.2 File Access

As an example we will develop our version of cat, a program, which concatenates a set of files from the hard-disk to print it to standard output (screen). E.g.

```
cat myfile.c  yourfile.c
```

prints both files in sequence on the screen.
Before a file can be read or written, it has to be opened with fopen(). fopen() returns a pointer to FILE for subsequent reads or writes on the file. The type name FILE is defined in <stdio.h>.

```
FILE  *fp;
FILE  *fopen(char *name, char *mode);
```

The call in a program is

```
fp = fopen(name, mode);
```

name is a char string containing the name of the file, mode is a char string indicating the use of the file:

"r" for read
"w" for write
"a" for append

If a file, that does not exist, is opened with "w" or "a", it is created.

Opening an existing file with "w" deletes the old content.

In case of error, *fopen()* returns NULL.

For simple reading and writing the contents of a file is just a stream of bytes, regardless of the meaning of the bytes.

Read is done with `int getc(FILE *fp);`
returning the next byte from the stream referred to by fp or EOF. fp is incremented.

Write is done with `int putc(int c, FILE *fp);`
writing the byte (character) c to the file fp.

When a 'C' program is started, the operating system provides 3 pointers called

stdin for standard input, normally keyboard
stdout for standard output, normally screen
stderr for standard error, normally screen

Formatted input and output of files is done in analogy to *printf()* and *scanf()* by

```
int fscanf (FILE  *fp, char  *format, ..........);
int fprintf(FILE  *fp, char  *format, ..........);
```

The function

```
int fclose(FILE *fp);
```

breaks the connection between the file pointer and the file, freeing fp for another file.

Now we can write our cat program

E.8-2

```
void  filecopy(FILE *infile, FILE *outfile);
```

```
int main(int argc, char *argv[])
/*************************************/
/*                                   */
/* cat: concatenate files            */
/*                                   */
/*************************************/
{
FILE  *fp;
int   i;

     if (argc == 1) { /* no args, copy stdin */
          filecopy(stdin, stdout);
     }
     else {
          i = 1;
          while( --argc > 0) {
               if ((fp = fopen(argv[i], "r")) == NULL) {
                    printf("can't open %s\n", *argv);
                    return -1;
               }
               else {
                    filecopy(fp, stdout);
                    fclose(fp);
                    i++;
               }
          }
     }
     return 0;

} /* END_main() */

void filecopy(FILE *infile, FILE *outfile)
/*************************************/
/*                                   */
/* copy file infile to file outfile.  */
/*                                   */
/*************************************/
{
int  c;

     while ((c = getc(infile)) != EOF) {
          putc(c, outfile);
     }
} /* END_filecopy() */
```

You can try

```
     cat E_2_2.c  E_2_4.c
```

You can also try

```
     cat cat.exe
```

The result is garbage.

E.8-3

The difference between *fprintf()* and *putc()*:

```
int main(void)
```

```
/*************************************/
/*                                   */
/* demonstrate fprintf()             */
/*                                   */
/*************************************/
{
FILE  *fp;
int   myint;

     fp = fopen("myfile", "w");
     myint = 6;
     fprintf(fp, "%d", myint);

     return 0;

} /* END_main() */
```

Using '>type myfile' or using an editor shows a 6 in myfile.
'>dir myfile' or using the windows explorer for myfile shows a size of 1 byte.

Now substitute the fprintf()-line by

```
fprintf(fp, "%d %6d", myint, 37);
```

Using '>type myfile' or using an editor shows a '6   37' in myfile.
'>dir myfile' or using the windows explorer for myfile shows a size of 8 bytes.

Now we substitute the body of main() by

```
int  i, myint;
char *cpointer;

     fp = fopen("myfile", "w");
     myint = 6;
     cpointer = (char *)&myint;

     for (i = 0; i < sizeof(myint); i++) {
          putc(*(cpointer++), fp);
     }


     return 0;

} /* END_main() */
```

Using '>type myfile' or using an editor shows a 'garbage' in myfile.
'>dir myfile' or using the windows explorer for myfile shows a size of 4 bytes.

We copied myint as a byte stream to the file.

8.3 Some String Operations and the system() function

We will demonstrate two of the many library functions for string operations:
Assume `char *s;`

```
          char *t;
          char c;

     strcmp(s,t);
```

compares two strings s and t and the return value indicates the lexicographic relation of s to t:
< 0, if s is less than t,
= 0, if s is identical to t,
> 0, if s is greater than t.

```
     strchr(s, c);
```

returns a pointer to the first character c, which appears in s, or NULL, if there is no such character.

E.8-4

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


int main(void)
/*************************************/
/*                                   */
/* strcmp() and strchr()             */
/*                                   */
/*************************************/
{
int result;
char *text[]= {"base string",
               "base string",
               "base nomore",
               "base string and more" };
char *pointer;

     result = strcmp(text[0],text[3]);
     printf("compare result %d\n", result); /* output is -1 */
                                            /* '\0' < ' '   */
     result = strcmp(text[0],text[2]);
     printf("compare result %d\n", result); /* output is +1 */
                                            /* 's' > 'n'    */
     result = strcmp(text[0],text[1]);
     printf("compare result %d\n", result); /* output is  0 */

     pointer = strchr(text[2], 's');
     printf("points to %c\n", *pointer);
     printf("next char is %c\n", *(++pointer));

     return 0;
} /* END_main() */
```

Now something else: We can start another program or a command from inside a 'C'-program using the *system()* function.

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

```
int system(const char *command);
```

E.8-5

```
system("dir");
getchar();
```

prints the directory list to the screen and waits for a key enter.

```
system("cat.exe E_2_2.c E_2_4.c");
```

starts cat to concatenate both files and print it on the screen.

---

9 Thread and Mutex

A *process* is an executing program. Threads are tasks, which are generated by a process and which depend on this process. A thread can use resources of this process. One or more threads run in the context of the process. A *thread* is the basic unit to which the operating system allocates processor time.

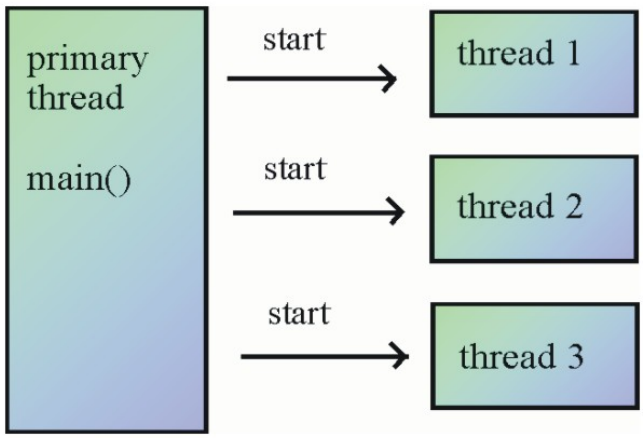
Programming a thread means to program a C-function, which is started by an API-call (*application programming interface)*. The C-function can call other C-functions. A thread is a set of C-functions. Threads are timely independent from each other and from the generating process (primary thread).

Each process provides the resources needed to execute a program. A process has a virtual address space, executable code, data, object handles, environment variables, a base priority. Each process is started with a single thread, often called the *primary thread*, and can create additional threads.

All threads of a process share its virtual address space and system resources. In addition, each thread maintains exception handlers, a scheduling priority, and a set of structures the operating system will use to save the thread context until it is scheduled. The *thread context* includes the thread's set of machine registers, a user stack and other entities.

Microsoft® Windows NT® supports *preemptive multitasking*, which creates the effect of simultaneous execution of multiple threads. On a multiprocessor computer, Windows NT can simultaneously execute as many threads as there are processors on the computer.

E.9-1



Pic.9-1

To work with threads, Windows 7 needs the multithreading version of the system library. Set the option for the compiler by:

->Project->Projectname Properties->Configuration Properties->C/C++->Codegeneration->
->Runtime library->Multi-threaded Debug DLL(/MDd)

We start a thread with the system call

```
_beginthread(void( __cdecl *start_address )( void * ),
             unsigned stack_size,
             void *arglist
             );
```

_beginthread needs `#include <process.h>`

start_address is the start address of a routine that begins execution of the new thread. As actual parameter we just pass the name of the start routine of our thread.

stack_size is the stack size for the new thread or 0. We will use 0 as actual parameter, in which case the operating system uses the same value as for the stack specified for the main thread.

arglist is the argument list to be passed to the new thread or NULL. Note: *arglist* is a void pointer.

If successful, _beginthread returns a handle to the newly created thread.

You can call

```
void _endthread( void );
```

explicitly to terminate a thread; however, *_endthread()* is called automatically when the thread returns from the routine. Terminating a thread with a call to *endthread()* helps to ensure proper recovery of resources allocated for the thread. *_endthread()* automatically closes the thread handle.

Now assume, we have defined a function *thread1(void *dummy)*, which shall print a line onto the screen.

E.9-2_start thread 1 in main()

```
int main()
/*****************************************/
/*                                       */
/* demonstrate effect of thread          */
/*                                       */
/*****************************************/
{
int looplimit = 5;

     /* start a thread                    */
     _beginthread( thread1, 0, &looplimit );

     Sleep(10000L);
     printf("close program with any key:\n");
     getchar();
} /* END_main() */


void thread1( void *dummy )
/*****************************************/
/*                                       */
/* thread prints several times his name  */
```

```
/*                                          */
/******************************************/

{
char output[]="thread1111111111111111111111111111111111111111";
int  i;
int  *ipoint;

     ipoint = (int *)dummy;
     for (i = 0; i < *ipoint; i++) {
          printf("my name %s\n", output);
     }
    _endthread();
} /* END_thread1() */
```

Next we write a function `void printslowly(char *whattoprint)` to print a line slowly char by char, that we can follow the printing on the screen.

To slow down the printing we use the function

```
void  Sleep(DWORD  milliseconds);
```

Sleep() lets the calling thread sleep for the time in milliseconds. DWORD is 32-bit unsigned integer.

E.9-3

```
void printslowly(char *whattoprint)
/************************************************/
/*                                              */
/* slow print to screen to make effects visible  */
/*                                              */
/************************************************/
{
char *control;
     control = whattoprint;

     while (*control != '\0') {
          printf("%c",*control);
          Sleep(10L); /* slow down printing. */
          control++;
     }
     printf("\n");
} /* END_printslowly() */
```

Let us add two other threads to our program. They will also print their thread number to the screen and they also call *printslowly()*. It is possible to access *printslowly()* quasi simultaneously, because *printslowly()* is *reentrant*. A computer program or routine is described as *reentrant* if it can be safely called again before its previous invocation has been completed, i.e. it can be safely executed concurrently. (Wikipedia). There is only one copy of the code in memory.

E.9-4 3 threads, the complete program:

```
// compile with: /MT /D "_X86_" /c??
// ->Project->Projectname Properties->Configuration Properties->C/C++
//->Codegeneration->->Runtime library->Multi-threaded Debug DLL(/MDd)

)
#include <windows.h>
#include <process.h>    /* _beginthread, _endthread */
#include <stddef.h>
#include <stdlib.h>

/*************** prototypes *********************/
void thread1    ( void *dummy );
void thread2    ( void *dummy );
void thread3    ( void *dummy );
void printslowly(char *whattoprint);

int main()
/****************************************/
/*                                      */
/* demonstrate effect of thread         */
/*                                      */
/****************************************/
{
int looplimit = 5;

      /* start 3 threads                 */
      _beginthread( thread1, 0, &looplimit );
      _beginthread( thread2, 0, &looplimit );
      _beginthread( thread3, 0, &looplimit );

      Sleep(10000L);
      printf("close program with any key:\n");
    getchar();
} /* END_main() */


void thread1( void *dummy )
/****************************************/
/*                                      */
/* thread prints several times his name */
/*                                      */
/****************************************/
{
char  output[] = "thread1111111111111111111111111111111111111111111111";
int   i;
int   *ipoint;

      ipoint = (int *)dummy;
      for (i = 0; i < *ipoint; i++) {
            printslowly(output);
      }
    _endthread();
} /* END_thread1() */

void thread2( void *dummy )
/****************************************/
/*                                      */
/* thread prints several times his name */
/*                                      */
/****************************************/
{
```

```
char  output[] = "thread2222222222222222222222222222222222222222";
int   i;
int   *ipoint;

      ipoint = (int *)dummy;
      for (i = 0; i < *ipoint; i++) {
            printslowly(output);
      }
    _endthread();
} /* END_thread2() */

void thread3( void *dummy )
/****************************************/
/*                                      */
/* thread prints several times his name   */
/*                                      */
/****************************************/
{
char  output[] = "thread3333333333333333333333333333333333333333";
int   i;
int   *ipoint;

      ipoint = (int *)dummy;
      for (i = 0; i < *ipoint; i++) {
            printslowly(output);
      }
    _endthread();
} /* END_thread3() */

void printslowly(char *whattoprint)
/************************************************/
/*                                              */
/* slow print to screen to make effects visible  */
/*                                              */
/************************************************/
{
char *control;
      control = whattoprint;

      while (*control != '\0') {
            printf("%c",*control);
            Sleep(10L); /* slow down printing. */
            control++;
      }
      printf("\n");
} /* END_printslowly() */
```

Running the program, we watch, that the output is mixed up. No thread is able to print a whole 'output[]'-line without being interrupted by another thread.

To enable a thread to print a complete line to the screen, we introduce the mutex concept. The MSDN library (Microsoft Developer Network) gives information about mutex objects, which are operating system resources.

A *mutex object* is a synchronization object whose state is set to signaled when it is *not* owned by any thread, and nonsignaled when it is owned. Only one thread at a time can own a mutex object, whose name comes from the fact that it is useful in coordinating mutually exclusive access to a shared resource. For example, to prevent two threads from writing/reading to the

same global array[] at the same time, each thread waits for ownership of a mutex object before executing the code that accesses the memory. After writing to the shared memory, the thread releases the mutex object.

A thread uses the *CreateMutex()* function to create a mutex object. The creating thread can request immediate ownership of the mutex object and can also specify a name for the mutex object.

Any thread with a handle to a mutex object can use a wait function to request ownership of the mutex object. If the mutex object is owned by another thread, the wait function blocks the requesting thread until the owning thread releases the mutex object using the *ReleaseMutex()* function.

If more than one thread is waiting on a mutex, a waiting thread is selected by the operating system. Do not assume a first-in, first-out (FIFO) order. External events can change the wait order.

Create a mutex object:

```
HANDLE hMutex;

hMutex = CreateMutex(
             LPSECURITY_ATTRIBUTES lpMutexAttributes,
             BOOL                  bInitialOwner,
             LPCTSTR               lpName            );
```

The types are defines in some Microsoft header files.

lpMutexAttributes is a Pointer to a SECURITY_ATTRIBUTES structure that determines whether the returned handle can be inherited by child processes. If *lpMutexAttributes* is NULL, the handle cannot be inherited. Note: a thread is not a child process. A child process is created by other means.

bInitialOwner :If this value is TRUE, the calling thread obtains initial ownership of the mutex object.

lpName is a pointer to a null-terminated string specifying the name of the mutex object.

The wait function:

```
DWORD WaitForSingleObject(HANDLE hHandle,
                          DWORD  dwMilliseconds );
```

hHandle is the handle to the object, which is our mutex.

dwMilliseconds is the time-out interval, in milliseconds. The function returns if the interval elapses, even if the object's state is nonsignaled. Remember

signalled ^=^ free
nonsignaled ^=^occupied

The *WaitForSingleObject()* function checks the current state of the specified object. If the object's state is nonsignaled, the calling thread enters the wait state. It uses no processor time while waiting for the object state to become signaled or the time-out interval to elapse.

Release a mutex:

```
BOOL ReleaseMutex(HANDLE hMutex);
```

hMutex is the handle to the mutex object.

Finally use the *CloseHandle()* function to close the handle. The mutex object is destroyed when its last handle has been closed.

```
BOOL CloseHandle(HANDLE hObject);
```

Note: Closing a thread handle does not terminate the associated thread!! To remove a thread object, you must terminate the thread, then close all handles to the thread.

E.9-5

```
// compile with: /MT /D "_X86_" /c??
// ->Project->Projectname Properties->Configuration Properties->C/C++
//->Codegeneration->->Runtime library->Multi-threaded Debug DLL(/MDd)
#include <windows.h>
#include <process.h>    /* _beginthread, _endthread */
#include <stddef.h>
#include <stdlib.h>

/*************** prototypes *********************/
void thread1( void *dummy );
void thread2( void *dummy );
void thread3( void *dummy );
void printslowly(char *whattoprint);

/************* globalvariable ******************/

HANDLE hMutex;

int main()
/****************************************/
/*                                      */
/* demonstrate effect of thread and mutex */
/*                                      */
/****************************************/
{
int looplimit = 5;

      // Create a mutex with no initial owner.
      hMutex = CreateMutex(
                        NULL,               // no security attributes
                        FALSE,              // initially not owned
                        (LPCTSTR)"MyMutex"); // name of mutex

      if (hMutex == NULL) {
            printf("Mutex Create Error\n");
            exit(-1);
      }
      /* start 3 threads                  */
```

```
      _beginthread( thread1, 0, &looplimit );
      _beginthread( thread2, 0, &looplimit );
      _beginthread( thread3, 0, &looplimit );

      Sleep(15000L);
      printf("close program with any key:\n");
      getchar();
      CloseHandle(hMutex);
} /* END_main() */


void thread1( void *dummy )
/*****************************************/
/*                                       */
/* thread prints several times his name  */
/*                                       */
/*****************************************/
{
char  output[] = "thread1111111111111111111111111111111111111111111111";
int   i;
int   *ipoint;
DWORD dwWaitResult;

      ipoint = (int *)dummy;
      for (i = 0; i < *ipoint; i++) {
            dwWaitResult = WaitForSingleObject(
                                 hMutex,  // handle to mutex
                                 60000L); // 60-second time-out interval
            printslowly(output);
            ReleaseMutex(hMutex);
      }
    _endthread();
} /* END_thread1() */

void thread2( void *dummy )
/*****************************************/
/*                                       */
/* thread prints several times his name  */
/*                                       */
/*****************************************/
{
char  output[] = "thread2222222222222222222222222222222222222222222222";
int   i;
int   *ipoint;
DWORD dwWaitResult;

      ipoint = (int *)dummy;
      for (i = 0; i < *ipoint; i++) {
            dwWaitResult = WaitForSingleObject(
                                 hMutex,  // handle to mutex
                                 60000L); // 60-second time-out interval
            printslowly(output);
            ReleaseMutex(hMutex);
      }
    _endthread();
} /* END_thread2() */

void thread3( void *dummy )
/*****************************************/
/*                                       */
/* thread prints several times his name  */
/*                                       */
```

```
/*****************************************/
{
char  output[] = "thread333333333333333333333333333333333333333333333";
int   i;
int   *ipoint;
DWORD dwWaitResult;

      ipoint = (int *)dummy;
      for (i = 0; i < *ipoint; i++) {
            dwWaitResult = WaitForSingleObject(
                                    hMutex,  // handle to mutex
                                    60000L); // 60-second time-out interval
            printslowly(output);
            ReleaseMutex(hMutex);
      }
      _endthread();
} /* END_thread3() */

void printslowly(char *whattoprint)
/*************************************************/
/*                                               */
/* slow print to screen to make effects visible  */
/*                                               */
/*************************************************/
{
char *control;
      control = whattoprint;

      while (*control != '\0') {
            printf("%c",*control);
            Sleep(10L); /* slow down printing. */
            control++;
      }
      printf("\n");
} /* END_printslowly() */
```

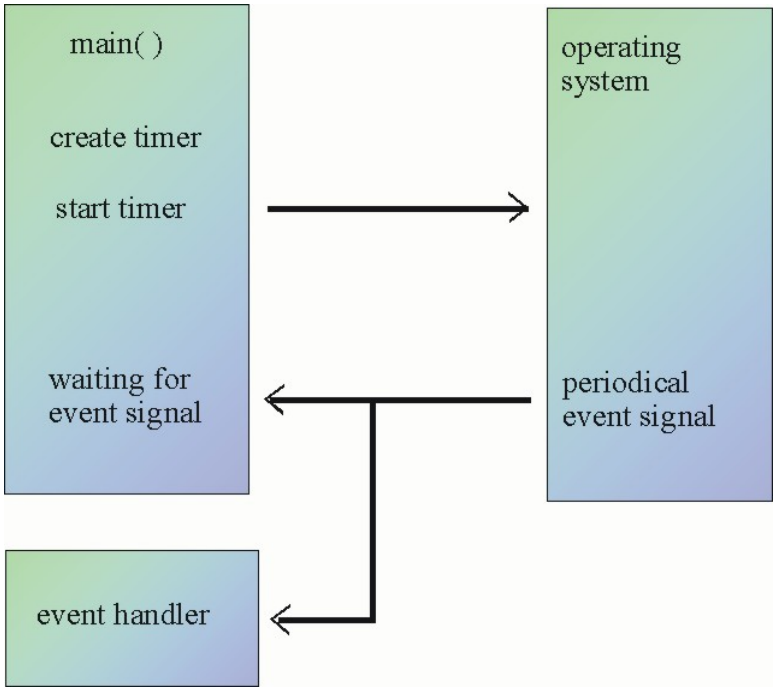
We didn't implement error handling for the calls of the system functions. For professional applications you should improve the program.

10 Timer in Windows Operating Systems

10.Timer in Windows Operating Systems

In this section we will deal with a periodical timer. This is a resource of the operating system which delivers a periodical event signal. We write an event handler (a function), which is called from the operating system as a reaction on the event.

A timer is first created by our *main()* process. Then the timer is started. After that we force our process to sleep until he notes the event signal, then the *main()* algorithm continues. In our examples *main()* acts as a second event handler, because *main()* is periodically waked up on the appearance of the event signal.



Pic.10-1

To realize a timer we need system functions from the library.

CreateWaitableTimer() creates the timer and returns a handle, which is an identification object for the timer (think of a positive number).

SetWaitableTimer () starts the timer. There is one parameter which tells the timer when to start from now, and another parameter defining the period.

SleepEx () forces the *main()* process to sleep until it receives the event signal.

CancelWaitableTimer () and *CloseHandle ()* hand back the operating system resource.

We explain the timer concept as far as we need to have an entry for future professional applications. For additional features see the literature.

In our first example we present furthermore an event handler, which reacts on <ctrl><c>. *SetConsoleCtrlHandler ()* redirects <ctrl><c> to our handler.

beep() generates a tone on the speaker.

E.10-1

```
/*************************************************************/
/* File TimeBeep.c                                           */
/* Literature Source http://world.std.com/~jmhart/timebeep.htm */
/*                                                           */
/* Version 2.1                                               */
/* 15.4.2010                                                 */
/* B. Guesmann                                               */
/*************************************************************/
/*                                                           */
/* Usage: TimeBeep Period.                                   */
/* Demonstration of a periodic alarm.                        */
/* This program uses a console control handler to catch      */
/* control-c signals.                                        */
/*                                                           */
/* This implementation uses the "waitable timers" introduced */
/* in NT 4.0 (also Windows 98) and supported by Visual C++.  */
/* The waitable timers are sharable named objects (their names share */
/* the same space used by synchronization and memory mapped objects).*/
/* The waitable timer is a synchronization object that you can wait on in */
/* another thread for the timer to signal. Alternatively, you can have a */
/* completion routine executed in the same thread by entering an */
/* "alertable wait state".                                   */
/*************************************************************/
#define _WIN32_WINNT 0x0400     /* Required in <winbase.h> to define
        waitable timer functions. This is not explained in the MS documentation.
        It indicates that you need NT 4.0. or later                */
#include <windows.h>
#include <winbase.h>
#include <stdio.h>

/****************** TimeBeep.c contains ************************/
int                 main      (int argc, LPTSTR argv []);
static BOOL WINAPI  Handler (DWORD CntrlEvent);
static VOID APIENTRY Beeper  (LPVOID, DWORD, DWORD);

/******************** Global Variables *************************/
volatile static BOOL Exit = FALSE;
HANDLE               hTimer;

int main (int argc, LPTSTR argv [])
/*********************************************************************/
/*                                                                   */
/* Starts Waitable Timer und waits as a Thread for the Signal of the Waitable Timer */
/*                                                                   */
/*********************************************************************/
{
DWORD                Count = 0, Period;
LARGE_INTEGER        DueTime;
```

```
        if (argc >= 2) {
                Period = atoi (argv [1]) * 1000;
        }
        else {
                printf("ERROR: Usage: TimeBeep period\n");
                exit(0);
        }

        if (!SetConsoleCtrlHandler (Handler, TRUE)) { /* redirect <strg><c>  */
                printf("Error setting event handler\n");
        }

        DueTime.QuadPart = -(LONGLONG)Period * 10000;
        /* Due time is negative for first time out relative to current time. Period is in ms (10**-3 sec)
        whereas the due time is in 100 ns (10**-7 sec) units to be consistent with a FILETIME. */

        hTimer = CreateWaitableTimer (
                        NULL,   /* Security attributes */
                        FALSE,  /* According to Microsoft Developer Network Documentation
                                   TRUE means, the timer is a manual-reset notification timer.
                                   Otherwise, the timer is a synchronization timer. Documentation
                                   does not explain the distinction.                          */
                        NULL    /* Do not name the timer - name space is shared with events,
                                   mutexes, semaphores, and mem map objects                   */ );
        if (hTimer == NULL) {
                printf("Failure creating waitable timer\n");
                exit(0);
        }

        if (!SetWaitableTimer (
                        hTimer,
                        &DueTime /* relative time of first signal */,
                        Period   /* Time period in ms */,
                        Beeper   /* Timer function, event handler */,
                        &Count   /* Parameter passed to timer function */,
                        TRUE     /* Does not apply - do not use it. */) )       {
                printf("Failure setting waitable timer\n");
                exit(0);
        }

         /* Enter the main loop */
        while (!Exit) {  /* Exit is changed by Handler() */
                printf ("Count = %d\n", Count);         /* Count is increased in the timer routine */
                SleepEx (INFINITE, TRUE);               /* Enter an alertable wait state. The timer
                                        handle is a synchronization object, this thread now waits on it. */
        }
        printf ("Shut down. Count = %d", Count);
        CancelWaitableTimer (hTimer);
        CloseHandle (hTimer);
        return 0;
} /* END_main() */
```

```
static VOID APIENTRY Beeper ( LPVOID lpCount,
                              DWORD dwTimerLowValue,
                              DWORD dwTimerHighValue  )
/*************************************************************************/
/* This is the event handler for the timer.                              */
/* The argument for this event handler is specified when the timer is made active,  */
/* in the lpCount parameter. The event handler also takes two DWORD values that     */
/* specify the high and low time values of the time at which the timer was signaled. */
/* These values are passed to the routine by the system using the FILETIME format.  */
/* For the definition of the prototype of this function see Microsoft Documentation. */
/*************************************************************************/
{
        *(LPDWORD)lpCount = *(LPDWORD)lpCount + 1;

        Beep (1000 /* Frequency */, 250 /* Duration (ms) */);
        return;
} /* END_Beeper() */

BOOL WINAPI Handler (DWORD CntrlEvent)
/*************************************/
/* reacts on <ctrl><c>               */
/*************************************/
{
        printf("Handler is called\n");
        Exit = TRUE;
        return TRUE;
} /* END_Handler() */
/* END_File_TimeBeep.c */
```

Another example:

E.10-2

```
/*************************************************************/
/* File WindowsTimer.c                                       */
/* Literature Source http://support.microsoft.com/kb/184796/de  */
/*              and      MSDN Library Visual Studio .Net     */
/* Version 2.1                                               */
/* 15.4.2010                                                 */
/* B. Guesmann                                               */
/*************************************************************/
/*                                                           */
/* Runs in a system window.                                  */
/* Demonstration of a periodic alarm.                        */
/*                                                           */
/* This implementation uses the "waitable timers" introduced */
/* in NT 4.0 (also Windows 98) and supported by Visual C++.  */
/* The waitable timers are sharable named objects (their names share */
/* the same space used by synchronization and memory mapped objects).*/
/* The waitable timer is a synchronization object that you can wait on in */
/* another thread for the timer to signal. Alternatively, you can have a */
/* completion routine executed in the same thread by entering an */
/* "alertable wait state".                                   */
/*************************************************************/
```

```
#define _WIN32_WINNT 0x0400
#include <windows.h>
#include <stdio.h>

#define _SECOND 10000000 /* 10**7 */

typedef struct _MYDATA {
        TCHAR *szText;
        DWORD dwValue;
} MYDATA;

/*************** WindowsTimer.c contains *********************/
int                     main            ( void );
VOID CALLBACK TimerAPCProc(LPVOID lpArg, DWORD dwTimerLowValue,
                                        DWORD  dwTimerHighValue );

int  main( void )
/*************************************************************/
/*                                                           */
/* creates a waitable timer and lets this thread wait for the timer signal.    */
/*                                                           */
/*************************************************************/
{
HANDLE              hTimer;
BOOL                bSuccess;
__int64             qwDueTime;
LARGE_INTEGER       liDueTime;
MYDATA              MyData;
char                szError[255];

        MyData.szText = "This is my data.";
        MyData.dwValue = 100;

        hTimer = CreateWaitableTimer(
                        NULL,                   // Default security attributes.
                        FALSE,                  // Create auto-reset timer.
                        (LPCTSTR)"MyTimer"  // Name of waitable timer.       );
        if (hTimer == NULL ) {
                wsprintf( szError, "CreateWaitableTimer() failed with Error %d.",GetLastError() );
                MessageBox( NULL, szError, "Error", MB_ICONEXCLAMATION );
                exit(0);
         }

        // Create a negative 64-bit integer that will be used to signal the timer 5 seconds from now.
        qwDueTime = -5 * _SECOND;

        // Copy the relative time into a LARGE_INTEGER.
        liDueTime.LowPart  = (DWORD) ( qwDueTime & 0xFFFFFFFF );
        liDueTime.HighPart = (LONG)  ( qwDueTime >> 32 );

        bSuccess = SetWaitableTimer(
                        hTimer,         // Handle to the timer object.
                        &liDueTime,     // When timer will become signaled.
                                                // (periodic time ticks start)
```

```
                                        // positive value means absolute time.
                                        // negative value means relative time.
                        2000,           // Periodic timer interval of 2 seconds.
                        TimerAPCProc,// Completion routine. Event handler
                        &MyData,        // Argument to the completion routine.
                        FALSE // Do not restore a suspended system.        );

        if ( bSuccess ) {
                for ( ; MyData.dwValue < 1000; MyData.dwValue += 100 ) {
                        /* wait 9 times to be awaked from the timer. */
                SleepEx(INFINITE,   // Wait forever.
                                TRUE );// IMPORTANT!!! The thread must be in an
                                        // alertable state to process the APC.
                }
        } else {
                wsprintf( szError, "SetWaitableTimer() failed with Error %d.", GetLastError() );
                MessageBox( NULL, szError, "Error", MB_ICONEXCLAMATION );
        }

        CloseHandle( hTimer );
        return 0;
} /* END_main() */

VOID CALLBACK TimerAPCProc(LPVOID lpArg,                    // Data value.
                                DWORD  dwTimerLowValue,   // Timer low value.
                                DWORD  dwTimerHighValue   // Timer high value. )
/*******************************************************/
/* Asynchronous procedure call as event handler.                */
/*******************************************************/
{
MYDATA *pMyData = (MYDATA *)lpArg;

        printf( "Message: %s\nValue: %d\n\n", pMyData->szText, pMyData->dwValue );
        MessageBeep(0);
} /* END_TimerAPCProc() */

/* END_File_WindowsTimer.c */
```

11 ASCII-Tabelle

Dr. B. Güsmann VGU

Introduction to the C Programming Language

ASCII

American Standard Code for Information Interchange

Dez	Hex	Okt		Dez	Hex	Okt		Dez	Hex	Okt		Dez	Hex	Okt	
0	0x00	0	NUL	32	0x20	040	SP	64	0x40	100	@	96	0x60	140	`
1	0x01	1	SOH	33	0x21	041	!	65	0x41	101	A	97	0x61	141	a
2	0x02	2	STX	34	0x22	042	"	66	0x42	102	B	98	0x62	142	b
3	0x03	3	ETX	35	0x23	043	#	67	0x43	103	C	99	0x63	143	c
4	0x04	4	EOT	36	0x24	044	$	68	0x44	104	D	100	0x64	144	d
5	0x05	5	ENQ	37	0x25	045	%	69	0x45	105	E	101	0x65	145	e
6	0x06	6	ACK	38	0x26	046	&	70	0x46	106	F	102	0x66	146	f
7	0x07	7	BEL	39	0x27	047	'	71	0x47	107	G	103	0x67	147	g
8	0x08	10	BS	40	0x28	050	(	72	0x48	110	H	104	0x68	150	h
9	0x09	11	TAB	41	0x29	051	)	73	0x49	111	I	105	0x69	151	i
10	0x0A	12	LF	42	0x2A	052	*	74	0x4A	112	J	106	0x6A	152	j
11	0x0B	13	VT	43	0x2B	053	+	75	0x4B	113	K	107	0x6B	153	k
12	0x0C	14	FF	44	0x2C	054	,	76	0x4C	114	L	108	0x6C	154	l
13	0x0D	15	CR	45	0x2D	055	-	77	0x4D	115	M	109	0x6D	155	m
14	0x0E	16	SO	46	0x2E	056	.	78	0x4E	116	N	110	0x6E	156	n
15	0x0F	17	SI	47	0x2F	057	/	79	0x4F	117	O	111	0x6F	157	o
16	0x10	20	DLE	48	0x30	060	0	80	0x50	120	P	112	0x70	160	p
17	0x11	21	DC1	49	0x31	061	1	81	0x51	121	Q	113	0x71	161	q
18	0x12	22	DC2	50	0x32	062	2	82	0x52	122	R	114	0x72	162	r
19	0x13	23	DC3	51	0x33	063	3	83	0x53	123	S	115	0x73	163	s
20	0x14	24	DC4	52	0x34	064	4	84	0x54	124	T	116	0x74	164	t
21	0x15	25	NAK	53	0x35	065	5	85	0x55	125	U	117	0x75	165	u
22	0x16	26	SYN	54	0x36	066	6	86	0x56	126	V	118	0x76	166	v
23	0x17	27	ETB	55	0x37	067	7	87	0x57	127	W	119	0x77	167	w
24	0x18	30	CAN	56	0x38	070	8	88	0x58	130	X	120	0x78	170	x
25	0x19	31	EM	57	0x39	071	9	89	0x59	131	Y	121	0x79	171	y
26	0x1A	32	SUB	58	0x3A	072	:	90	0x5A	132	Z	122	0x7A	172	z
27	0x1B	33	ESC	59	0x3B	073	;	91	0x5B	133	[	123	0x7B	173	{
28	0x1C	34	FS	60	0x3C	074	<	92	0x5C	134	\	124	0x7C	174	|
29	0x1D	35	GS	61	0x3D	075	=	93	0x5D	135	]	125	0x7D	175	}
30	0x1E	36	RS	62	0x3E	076	>	94	0x5E	136	^	126	0x7E	176	~
31	0x1F	37	US	63	0x3F	077	?	95	0x5F	137	_	127	0x7F	177	DEL

Links for the meaning of characters (text german with english definitions)

http://www.asciitable.de/tabelle.html

http://www.schatenseite.de/ascii-tabelle.html

12 References

(1) Kernighan,B.,W.,Ritchie,D.,M., The C Programming Language, Second Edition, Prentice Hall PTR, 2009
First Edition : The C Programming Language, Prentice Hall, 1978.

(2) Wikibook C
http://en.wikibooks.org/wiki/C_Programming

(3) The handbook for the GNU C Library:
http://www.gnu.org/software/libc/manual

(4) Frost. J.,Windows Sockets: A Quick And Dirty Primer
http://burks.bton.ac.uk/burks/pcinfo/progdocs/wsocktut/winsock.htm

(5) Microsoft Developer Network
http://msdn.microsoft.com/de-de/default.aspx

(6) Bauer, F., L., Der (ungerade) Collatz-Baum, Informatik Spektrum, Band 31, Heft 2, 2008

(7) Schröder-Preikschat, W., Lohmann, D., Scheler, F., Spinczyk, O., Dimensions of variability in embedded systems, Informatik-Forschung und Entwicklung, Band 22, Heft 1, 2007, pp. 5-22

(8) Bauer, A., Broy, M., Romberg, J., Schätz, B., Braun, P., Freund, U., Mata, N., Sandner, R., Mai, P., Ziegenbein, D., Das AutoMoDe-Projekt, Informatik-Forschung und Entwicklung, Band 22, Heft 1, 2007, pp. 45-57