Lecture 13 - Some additional features of C

Pointers to functions

```
/* Numeric and lexicographic sorting */
#include <stdio.h>
#include <string.h>

#define MAXLINES 5000 /* maximal number of lines
                         to sort */
char *lineptr[MAXLINES];  /* pointers to text
                             lines */

int readlines(char *lineptr[], int nlines);
void writelines(char *lineptr[], int nlines);

void qqsort(void *lineptr[], int left, int right,
            int (*comp)(void *, void *));
int numcmp(const char *, const char *);

/* sort lines from the input */
main(int argc, char *argv[])
{
  int nlines; /* number of all lines */
  int numeric=0; /* 1 - if numeric sorting */

  if (argc > 1 && strcmp(argv[1], "-n") == 0)
    numeric=1;
  if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
    qqsort((void **) lineptr, 0, nlines-1,
          (int (*) (void *, void *))
          (numeric ? numcmp : strcmp));
    writelines(lineptr, nlines);
    return 0;
  } else {
    printf("too many lines to sort \n");
    return 1;
  }
}

/* qsort: sort v[left] ... v[right] increasingly */
void qqsort(void *v[], int left, int right,
           int (*comp)(void *, void *))
{
  int i,last;
  void swap(void *v[], int, int);
  if (left >= right) /* do nothing if array */
    return;     /* contains fewer than two elements */
  swap(v, left, (left+right)/2);
  last=left;
  for (i=left+1; i<=right; i++)
    if ((*comp)(v[i],v[left]) < 0)
      swap(v, ++last, i);
  swap(v, left, last);
  qqsort(v, left, last-1, comp);
  qqsort(v, last+1, right, comp);
}

#include <stdlib.h>

/* numcmp: compare numerically s1 and s2 */
int numcmp(const char *s1, const char *s2)
{
  double v1,v2;

  v1=atof(s1);
  v2=atof(s2);
  if (v1 < v2)
    return -1;
  else if (v1 > v2)
    return 1;
  else
    return 0;
}
```

```
void swap(void *v[], int i, int j)
{
  void *temp;

  temp=v[i];
  v[i]=v[j];
  v[j]=temp;
}

#define MAXLEN 1000 /* length of the input line */
int getline(char *, int);

/* readlines: read lines from the input */
int readlines(char *lineptr[], int maxlines)
{
  int len, nlines=0;
  char *p, line[MAXLEN];

  while ((len=getline(line, MAXLEN)) > 0)
    if (nlines >= maxlines || (p=(char *)
                malloc(len))==NULL)
      return -1;
    else {
      line[len-1]='\0'; /* remove new line character */
      strcpy(p,line);
      lineptr[nlines++]=p;
    }
  return nlines;
}

/* writelines: print out lines to output */
void writelines(char *lineptr[], int nlines)
{
  int i;
  for (i=0; i < nlines; i++)
    printf("%s\n", lineptr[i]);
}

/* getline: read line to s, give its length */
int getline(char s[], int lim)
{
  int c,i;
  for (i=0; i<lim -1 && (c=getchar()) != EOF
                     && c != '\n'; ++i)
    s[i]=c;
  if (c=='\n')
    s[i]=c;
  else
    ++i;
  s[i]='\0';
  return i;
}
```

Pointer arguments of *void ** type

Function **qqsort** has got arguments of *void ** type for pointer arguments. By means of the casting operation any pointer may be converted to the *void ** and conversely without any loss of information.

Declaration

```
int (*comp) (void *, void *)
```

declares pointer to function returning integer value. But the following declaration

```
int *comp(void *, void *)
```

says that *comp* is the function returning pointer to objects of integer type.

Complicated declarations

Examples:

char **argv	argv: pointer to a pointer to char
int (*daytab)[13]	daytab: pointer to an array[13] with int elements
int *daytab[13]	daytab: array[13] with elements being pointers to int
void *comp()	comp: function returning pointer to void
void (*comp)()	comp: pointer to function returning void
char (*(*x())[])()	x: function returning pointer to array[] with elements pointer to function returning char
char (*(*x[3])())[5]	x: array[3] with elements pointer to function returning pointer to array[5] with elements char

Simplified form of grammatic syntax rules introducing declarators:

```
   declarator:
          optional * immediate-declarator

  immediate-declarator:
          name
          (declarator)
          immediate-declarator()
          immediate-declarator[optional size]
```

Example of program decoding simple declarations):

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define MAXTOKEN 100

enum ( NAME, PARENS, BRACKETS };

void dcl(void);
void dirdcl(void);
int gettoken(void);
int tokentype;        /* type of the last element */
char token[MAXTOKEN]; /* text of the last element */
char name[MAXTOKEN];  /* name aapearing in declar. */
char datatype[MAXTOKEN]; /*data type: char,int,etc.*/
char out[1000];       /* output verbal description */

main()  /* dcl: replace C declarations
           with verbal description */
{
  while (gettoken() != EOF) {/* type is the first */
    strcpy(datatype, token)  /* lexem in data */
    out[0]='\0';             /* line   */
    dcl();    /* syntax analysis of
                 the rest of line */
    if (tokentype != '\n')
      printf("syntax error\n");
    printf("%s: %s %s\n", name, out, datatype);
  }
  return 0;
}

int gettoken(void) /* give the next lexem */
{
  int c, getch(void);
  void ungetch(int );

  while ((c=getch()) == ' ' || c == '\t')
    ;
  if (c == '(') {
    if ((c=getch()) == ')') {
      strcpy(token, "()");
      return tokentype = PARENS;
    } else {
      ungetch(c);
      return tokentype = '(';
    }
  } else if (c == '[') {
      for (*p++ = c; (*p++ = getch()) != ']';)
        ;
      *p='\0';
      return tokentype = BRACKETS;
  } else if (isalpha(c)) {
      for (*p++ = c; isalnum(c=getch()); )
        *p++=c;
      *p='\0';
      ungetch(c);
      return tokentype = NAME;
  } else
      return tokentype = c;
}

#define BUFSIZE 100  /* max. buffer size */

char buf[BUFSIZE]; /* buffer for returns from ungetch*/
int bufp=0;  /* next free space in the buffer */

int getch(void) /* read character; may be returned */
{               /*  to the input */
  return (bufp > 0) ? buf[--bufp]:getchar();
}

void ungetch(int c) /* return character back to */
{                   /* the input */
  if (bufp >= BUFSIZE)
    printf("ungetch: too many returns!\n");
  else
    buf[bufp++]=c;
}


/* dcl: syntax declarator analysis */
void dcl(void)
{
  int ns;
  for (ns=0; gettoken()=='*';) /* counts *-s */
    ns++;
  dirdcl();
  while (ns-- > 0)
    strcat(out, "pointer  to");
}

/* dirdcl: syntax analysis of the immediate
                               declarator */
void dirdcl(void)
{
  int type;
  if (tokentype == '(') {  /* (declarator) */
    dcl();
    if (tokentype != ')')
      printf("Error: missing paranthesis )\n");
  } else if (tokentype == NAME) /* name of variable */
         strcpy(name, token);
  else
    printf("Error: expected  name  or"
                              " (declarator)\n");
  while (type=gettoken())==PARENS || type==BRACKETS)
    if (type == PARENS)   /* pair of paranthesis () */
      strcat(out,"function returning");
    else {        /* pair of paranthesis [] */
      strcat(out, " array");
      strcat(out,token);  /* ev. size */
      strcat(out," with elements");
    }
}
```

Variable length of arguments list

```
#include <stdio.h>
#include <stdarg.h>
/* minprintf: minimal printf with variable
number of arguments */
void minprintf(char *fmt, ... )
{
  va_list ap; /* points subsequently each not fixed */
  char *p, *sval;           /*  argument */
  int ival;
  double dval;
  va_start(ap, fmt); /* ap points first not fixed
   argument;   fmt - last fixed argument */
  for (p=fmt; *p; p++) {
    if (*p != '%') {
      putchar(*p);  continue;
    }
    switch(*++p) {
    case 'd':
      ival = va_arg(ap, int);
      printf("%d",ival);
      break;
    case 'f':
      dval=va_arg(ap, double);
      printf("%f",dval);
      break;
    case 's':
      for (sval=va_arg(ap, char *); *sval; sval++)
        putchar(*sval);
      break;
    default:
      putchar(*p);
      break;
    }
  }
  va_end(ap); /* finishing job, clean */
}
```

Three macrosses defined in `<stdarg.h>`:

```
va_start(va_list ap, lastfix);
type va_arg(va_list ap, type);
void va_end(va_list ap);

va_list   -  array storing information needed by
             va_arg and va_end
```

`va_arg` and `va_start` ensure a portable way to access variable lists of arguments.

- `va_start` sets pointer to the first not fixed argument passed to the function
- `va_arg` expands itself to an expression having the same type and value as the next argument passed
- `va_end` aids the function to perform normal return

Example (summation of numbers):

```
#include <stdio.h>
#include <stdarg.h>
/* calculate sum of a 0 terminated list */
void sum(char *msg, ...)
{
   int total = 0;
   va_list ap;
   int arg;
   va_start(ap, msg);
   while ((arg = va_arg(ap,int)) != 0) {
      total += arg;
   }
   printf(msg, total);
   va_end(ap);
}
int main(void) {
   sum("The total of 1+2+3+4 is %d\n", 1,2,3,4,0);
   return 0;
}
```

Enumeration constant
(List of constant integer values)

```
enum boolean {NO,YES};
enum escapes {BELL='\a', BACKSPACE='\b', TAB='\t',
              NEWLINE='\n', VTAB='\v', RETURN='\r'};

enum months {STY=1, LUT, MAR, KWI, MAJ, CZE, LIP,
             SIE, WRZ, PAZ, LIS, GRU};
   /*  months: february is second, march third etc. */
```

Names of different enumerations must be different. Within the same enumeration values may be repeated.

Name of *enum* type shares the same space in memory as names of structure and unions compositions.

Names of the enum variables belong to the same class as identifiers of ordinary variables.