Lecture 12 - Parametrization of programs

Three different mechanisms of parametrization:

- by PREPROCESOR directives
- main() function arguments
- *typedef* declaration

typedef declaration

Used to create new data types names.
Example I:

```
typedef int Length;  /*  Length  */
```

Further in declarations and castings we may use type *Length* as well as type *int*

```
Length len, maxlen;
Length *lengths[];
```

Example II (declaration):

```
typedef char *String; /* Text  */
```

usage:

```
String p, lineptr[MAXLINES], alloc(int);
int strcmp( String, String );
p=(String) malloc(200);
```

Example III:

```
typedef struct tnode { /* tree node */
  char *word;    /* pointer to the word text */
  int count;     /* appearance counter */
  struct tnode *left;  /* left child  */
  struct tnode *right; /* right child */
} Treenode, *Treeptr;
/* Treenode - node,
   Treeptr - pointer to node */
```

Both declarations form two new key words for type description: *Treenode* (structure) and *Treeptr* (pointer to such structure).

Functions assigning memory to new node may be written for instance in such way:

```
Treeptr talloc(void)
{
  return (Treeptr) malloc(sizeof(Treenode))
}
```

Remark: *Typedef* does not create new type; assigns only a new name to an already existing type:

Hence it is similar to the preprocessor directive
`#define`
however it is interpreted by the compiler.

Other examples:

```
1)  typedef int (*PFI)(); /* creates type PFI
     as a pointer to a function returning
     integer value  */

2)  typedef char Arr[3];
    Arr Vec, *Ptr;
    /* we have declared type Arr equaling it with
       type  "char  [3]"
       and consecutively array  Ver
                of type "char  [3]"
       and variable  Ptr of type "char (*) [3]" */

3)  typedef int *REF;
    REF Ptr, Arr[2];

4)  typedef struct CPLX {
      double Re, Im;
    } COMPLEX;
    COMPLEX Vec[3]; /* array of complex type */

5)  typedef float *Vec[3], (*REF) (long,int);
    Vec Arr[2];
    REF *Ptr; /* type "float (**) (long,int)" */
```

Two main reasons justifying the use of *typedef*:

1. parametrization of the program making easy transportation to other machines
2. better commenting of the program

Arguments of commands line

```
  echo ahoj, adventure

  copy <source file> <destination file>
```

Example program *echo*

I - we are treating **argv** as an array of pointers to character strings:

```
#include <stdio.h>

/*  echo of the call arguments; version 1  */
main(int argc, char *argv[])
{
  int i;
  for (i=1; i<argc; i++)
    printf("%s%s",argv[i], (i<argc-1) ? " ":"");
  printf("\n");
  return 0;
}
```

II - we are treating **argv** as a pointer to the pointer to characters string:

```
#include <stdio.h>

/*  echo of the call arguments: version 2  */
main(int argc, char *argv[])
{
  while (--argc > 0)
    printf("%s%s",*++argv, (argc>1) ? " ":"");
  printf("\n");
  return 0;
}
```

General structure of the main() function

```
/*  Preprocessor statements  */
/*  global statements        */

int main([[[ <arguments counter>], <list
 of arguments>], <list of environment
    arguments>])
int <arguments counter>;
char * <list of arguments>[];
char * <list of environment arguments>[];
{
   /*  main function statements  */
}
```

Remark: According to the convention:

- argv[0] points the name under which the program has been called.
- therefore argc should be greater than or equal 1.
- furthermore standard requires *argv[argc]* to be an empty pointer

Standard ANSI allows three optional arguments. According to the convention arguments should have the following names:

Name	Meaning
argc	arguments counter of integer type
argv	list of arguments in the form of strings
env	list of environment arguments in the form of strings

Using conventional names structure of the *main()* function may be written as follows:

```
int main([[[argc],argv],env])
int argc;
char *argv[];
char *env[];
{
  /* staements of the  main()  function */
}
```

According to the present standard the *main()* function may appear in four formats:

1. Without arguments

```
main(void)
{
  /*  statements of main() function */
}
```

2. with one argument (seldom used)

```
 main( int argc )
 {
   /*  statements of main() function */
 }
```

3. with two arguments (most frequently used)

```
main(int argc, char *argv[])
{
  /*  statements of main() function */
}
```

4. with three arguments:

```
main(int argc, char *argv[], char *env[])
{
  /*  statements of main() function */
}
```

Example I:

```
/* argc.c */
#include <stdio.h>
main(argc)
int argc;
{
 printf("Number of arguments of  "
        "the command line:   %d\n",argc);
}
```

Example II:

```
/* argv.c */
#include <stdio.h>
main(argc,argv)
int argc;
char *argv[];
{
 int count = 0;
 void newline (void);
 printf("List of arguments of the command line: \n\n");
 for (count = 0; count < argc ; count++ )
     printf("%s \n",argv[count]);
 newline();
}

void newline(void)
{
 printf("\n");
}
```

Example III:

```
/* env.c */
#include <stdio.h>
main(argc,argv,env)
int argc;
char *argv[];
char *env[];
{
 int count = 0;
 void newline (void);
 printf("List of the environment arguments "
        "of the command line:\n\n");
 for (count = 0; env[count] != 0; count++ )
    printf("%s\n",env[count]);
 newline();
}

void newline(void)
{
 printf("\n");
}
```

Example IV (search for a pattern):

```
#include <stdio.h>
#include <string.h>
#define MAXLINE 1000

int getline(char s[], int lim);
main(int argc, char *argv[])
{
   char line[MAXLINE];
   int found=0;
   if (argc != 2)
     printf("Usage:  find pattern\n");
```

```
   else
     while ( getline(line, MAXLINE) > 0)
       if (strstr(line, argv[1]) != NULL){
         printf("%s",line);
         found++;
       }
   return found;
}

/* getline: read line to array  s;
   give its length  */
int getline(char s[], int lim)
{
   int c,i=0;

   while (--lim>0 && (c=getchar())!=EOF && c!='\n')
     s[i++]=c;
   if (c=='\n')
     s[i++]=c;
   s[i]='\0';
   return i;
}
```

Example V (search for a pattern with options **-x** (printing lines that do not contain the pattern) **-n** (pointing the number of the printed line):

```
#include <stdio.h>
#include <string.h>
#define MAXLINE 1000

int getline(char s[], int lim);

/* find: print lines matching the pattern from
the first, obligatory argument  */
main(int argc, char *argv[])
{
   char line[MAXLINE];
   long lineno=0;
   int c, except=0, number=0, found=0;

   while (--argc > 0 && (*++argv)[0] == '-')
     while (c=*++argv[0])
       switch (c) {
       case 'x':
         except=1;
         break;
       case 'n':
         number=1;
         break;
       default:
         printf("find: unknown option %c\n",c);
         argc=0;
         found=-1;
         break;
       }
   if (argc != 1)
     printf("Usage: find -x -n pattern\n");
   else
     while (getline(line,MAXLINE) > 0) {
       lineno++;
       if ((strstr(line, *argv) != NULL) != except) {
         if (number)
           printf("%ld:",lineno);
         printf("%s",line);
         found++;
       }
     }
   return found;
}

/* getline: reads line to the array  s;
   gives its length    */
int getline(char s[], int lim)
{
   int c,i=0;

   while (--lim>0 && (c=getchar())!=EOF && c!='\n')
     s[i++]=c;
   if (c=='\n')
     s[i++]=c;
   s[i]='\0';
   return i;
}
```

Example VI (passing name of the file to open as the program argument):

```
#include <stdio.h>
int main(argc, argv)
int argc;
char *argv[];
{
 FILE *fileptr;
 if (( fileptr = fopen (argv[1], "rb" )) == NULL )
   {
     printf( "File  %s doesn't exist\n", argv[1] );
     return(1);
   }
 else
   printf( " File %s opened correctly\n", argv[1] );
fclose(fileptr);
return(0);
}
```