Lecture 5 - Functions and the program structure

Format of the function definition

```
return-type name-of-function(parameters declarations)
  {
    declarations and statements
  }
```

```
The simplest function

dummy() {}
```

The *return* statement - tool to transfer value of a certain expression to the calling place.

```
return expression;
```

If it is necessary the returned expression is converted to the type of value returned by the function. Sometimes the returned expression is surrounded by the braces, although it is not necessary.

Functions returning noninteger values

- Definition of a function - example

```
#include <ctype.h>

/* conversion of the string  s
to a floating-point number */
double atof(char s[])
{
  double val, power;
  int i,sign;
  for (i=0; isspace(s[i]); i++)
    ;  /* We are omitting white characters */
  sign=(s[i] == '-') ? -1:1;
  if (s[i] == '+' || s[i] == '-')
    i++;
  for (val=0.0; isdigit(s[i]); i++)
    val = 10.0*val +(s[i]-'0');
  if (s[i]=='.')
    i++;
  for (power=1.0; isdigit(s[i]); i++) {
    val=10.0*val + (s[i]-'0');
    power*=10.0;
  }
  return sign*val\power;
}
```

- Function declaration

```
#include <stdio.h>
#define MAXLINE 100
/*  summing up the entered numbers  */
main()
{
  double sum=0.0, atof(char []);
  char line[MAXLINE];
  while (gets(line)>0) {
    if (line[0] == '\0') break;
    printf("\t%g\n",sum+=atof(line));
  }
  return 0;
}
```

External variables

- **inner variables** - arguments and variables defined inside the function
- **external variables** - variables defined outside all functions. They exist also after quitting a function.

Scope of names

- The scope of an automati8c variable declared at the beginning of a function is the whole body of the function containing the variable declaration.
- Local variables appearing in different functions are not in any sense interconnected. Function parameters are also local variables.
- The scope of an external variable and external function reaches from the place of its declaration in the file till the end of that file.
- If an external variable is used or an external function is called before their definitions or their definition is placed in other source file than the call then it is obligatory to use the **extern** declaration.

Example (What will be the result?):

```
#include <stdio.h>
int n=10, q=2;

void main()
{
  int fun(int);
  void f(void);
  int n=0, p=5;
  n=fun(p);
  printf("A: w main, n=%d, p=%d, q=%d\n",n,p,q);
  f();
}

int fun(int p)
{
  int q;
  q=2*p+n;
  printf ("B: w fun,  n=%d, p=%d, q=%d\n", n,p,q);
  return q;
}

void f(void)
{
  int p=q*n;
  printf ("C: w f,    n=%d, p=%d, q=%d\n", n,p,q);
}

Solution:
B: w fun,  n=10, p=5, q=20
A: w main, n=20, p=5, q=2
c: w f,    n=10, p=20, q=2
```

Header files

Example (program divided between two files):

```
 /*   File 1  */
#include <stdio.h>
#include "defs.h"

int n=10, q=2;

void main()
```

```
{
  int n=0, p=5;
  n=fun(p);
  printf("A: w main, n=%d, p=%d, q=%d\n",n,p,q);
  f();
}
```

```
  /*  File 2  */
#include <stdio.h>
extern int n,q;

int fun(int p)
{
  int q;
  q=2*p+n;
  printf ("B: w fun,  n=%d, p=%d, q=%d\n", n,p,q);
  return q;
}

void f(void)
{
  int p=q*n;
  printf ("C: w f,    n=%d, p=%d, q=%d\n", n,p,q);
}
```

```
  /*  File defs.h  */
extern int fun(int);
extern void f(void);
```

Static variables

static declaration applied to external variables and functions limits their scope from the place of their appearance till the end of the file. Therefore it is the way to hide their names. They will not be recognized in other files containing the rest of the program.

```
static char buf[BUFSIZE];
static int bufp=0;
```

static declaration with respect to the internal variables causes that they do not disappear between the subsequent function calls. They still remain local.

Register varaiables

register declaration informs the compiler that the variable will be intensively used. Such variables compiler may placed directly in the machine registers.

Possible use – exclusively with respect to automatic variables and formal function parameters.

```
register int x;
register char c;

f( register unsigned m, register long n)
{
  register int i;
  ....
}
```

There doesn't exist any possibility to get the address of a register variable.

The block structure

- It is possible to declare variables inside a block (compound statement).
- It is forbidden to define function inside another function.
- Automatic variables inside a function (together with their parameters) take precedence before the external variables and functions of the same name.

```
if (n>0) {
  int i; /* definition of a new  i */
  for (i=0; i<n; i++)
    .....
}
/* The scope of i is the "true" branch of the  if */
```

Initialization

- If the initial values are not given, then the external and static variables are initialized with zeros.
- Initial values of the automatic and register variables are not defined.
- Initial value of an external or static variable must be a constant expression.
- Automatic and register variables are initialized after each entrance to the function or block.

```
Array declaration:

int days[]={31,28,31,30,31,30,31,31,30,31,30,31};

For character arrays we may use a string constant:

char patt[]="grass";
is equivalent to:
char patt[]={'g','r','a','s','s','\0'};
```

Reccursion

Call of a function inside the same function.

Example 1:

```
/*   1  */
int silnia(int n)
{
   int wynik=1;
   while (n--)
     wynik*=n;
   return wynik;
}

/*  2  */
int silnia(int n)
{
   if (n == 1)
     return 1;
   else
     return n*silnia(n-1);
}
```

Preparing and running a complete C Program

Planning a program

"Top-down" programming.

The overall progarm strategy should be completely mapped out before any of the detailed programming actually begins. This entire process may be repeated several times, with more programming details added at each stage. When the overall strategy is set then the syntactic details of the language may be considered.

Top-down program organization is normally carried out by developing an informal outline consisting of phrases and sentences written partly in english partly in C. This is followed by the so called pseudocode.

Example

Determine how much money will accumulate in bank account after n years if a known amount P is deposited initially and the interest rate is r percent per year, compounded annually.

The program general outline is as follows:

1. Declare the required program variables.
2. Read in values for the principal P, the interest rate r and the number of years n.
3. Calculate the the decimal representation of the interest rate i using the formula

$$i = \frac{r}{100.0}$$

4. Determine the future accumulation F using the formula

$$F = P(1+i)^n$$

5. Write out the calculated value for F

```
/*  compound interest calculations  */

main()
{
   /*  declare the program variables  */

   /*  read in values for P, r and n  */

   /*  calculate value for i    */

   /*  calculate value for F    */

   /*  write out the calculated value for  F  */
}
```

A more detailed version of the above outline

```
/*  compound interest calculations  */

main()
{
   /*  p, r, n, i  and  f  to be
             floating point variables  */

   /*  write a prompt for p and then read its value  */
   /*  write a prompt for r and then read its value  */
   /*  write a prompt for n and then read its value  */

   /*  calculate  i=r/100.0  */
                           n
   /*  calculate f = p(1+i)   as follows

       f = p * pow((1+i),n)
       where  pow  is a library function
       for exponentiation   */

   /*  write the value for  f,
   with an accompanying label  */
}
```

Another method sometimes used when planning a C program is the "bottom-up" approach. It involves the detailed development of the self-contained program modules early in the overall planning process. The overall progarm development is then based upon the characteristics of these available progarm modules.

Writing a C program

Inclusion of certain additional features is a good practice:

1. logical sequencing of the statements
2. the use of indentation (illustrates the subordinate nature of individual statements within a group)
3. comments (if written properly they provide a useful overview of the general program logic).
4. ability to generate clear, legible output (Two factors contribute: i) labelling the output data ii) appearance of some of the input data together with the output to allow an identification of the current program execution).
5. an interactive program should generate prompts at appropriate times during the program execution in order to provide the use an information, how to input the data.

Example compound interest sample program

```
#include <stdio.h>
#include <math.h>
/*  simple compound interest problem  */
void main()
{
   float p,r,n,i,f;

   /*  read input data (include prompts)  */
   printf("Please enter a value "
          "for the principal (P): ");
   scanf("%f", &p);
   printf("Please enter a value "
          "for the interest rate (r): ");
   scanf("%f", &r);
   printf("Please enter a value "
         "for the number of years (n): ");
   scanf("%f", &n);

   /*  calculate i, then  f  */
   i=r/100;
   f=p * pow((1+i), n);

   /*  write output  */
   printf(:\nThe final value (F) is: %,2f\n",f);
}
```

Entering the program into the computer

Use an editor to enter the program into a text file line-by-line. The suffix *c* should be attached to he file name which identifies the file as a C program.

Compiling and executing the program

- compilation
- linking
- executing

Error diagnostics

- syntactic (grammatical) errors

  Example

```
#include <stdio.h>
include <math.h>
/*  simple compound interest problem  */
void main
{
   float p,r,n,i;

   /*  read input data (include prompts)  */
   printf("Please enter a value "
          "for the principal (P): ");
   scanf("%f", &p);
   printf("Please enter a value "
          "for the interest rate (r): );
   scanf("%f", &r);
   printf("Please enter a value "
          "for the number of years (n): ");
   scanf("%f", n)

   /*  calculate i, then  f  */
   i=r/100
   f=p * pow(1+i), n);

   /*  write output  *
   printf(:\nThe final value (F) is: %,2f\n",f);
```

  The errors are as follows:

  1. The second *include* statement does not begin with a #-sign; *main* does not include a pair of parantheses.
  2. The variable f is not declared to be a floating-point variable.
  3. the control string in the second printf statement does not have a closing quotation mark.
  4. The last *scanf* statement and the assignement for i do not end with semicolons.
  5. The assignement statement for f contains unbalanced parantheses.
  6. The last comment lacks the final slash ( ).
  7. The program does not end with a closing brace (}).

- execution errors

  Example - real roots of a quadratic form

  Calculate the real roots of the quadratic equation

$$ax^2 + bx + c = 0$$

  using the quadratic formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

  Below is a sample C program

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

/*  real roots of a quadratic equation  */
void main()
{
   float a,b,c,d,x1,x2;

   /*  read input data  */
   printf("a = ");
   scanf("%f", &a);
   printf("b = ");
   scanf("%f", &b);
   printf("c = ");
   scanf("%f", &c);

   /*  carry out the calculations  */
   d=sqrt(b*b - 4*a*c);
   x1=(-b+d)/(2*a);
   x2=(-b-d)/(2*a);

   /*  write output  */
   printf("x1 = %e   x2 = %e", x1, x2);
}
```

Logical debugging

- Detecting errors

  The first step is to test a new program with data that will yield a known answer.

- Correcting errors

  Very seldom the source of a subtle error lies in the hardware or compiler. Usually it is an error in the program logic.