Lecture 6 - Arrays and pointers

Pointers and addresses

x - variable
px = &x; - assignment of the address of variable x to variable px
px=x; y=px; is equivalent to y=x;

Declarations: int x,y;
int *px;
float *pz;

Pointer should indicate an object of a given type!!
Exclusion - pointer to void !!!!

Equivalent notations:
*ip+=1; ++*ip; (*ip)++;

In the third case the brackets are necessary, since operations determined by one argument * and ++ are executed from the right to the left.

Pointers and arrays

int a[10];
int *pa,x;
pa=&a[0]; x=*pa; is equivalent to x=a[0];
*(pa+1) refers to a[1]
*p+1 address to a[1]
*(p+1) is equal to a[1]
array name == pointer to the zero element of the array
a == &a[0] hence
pa=&a[0]; is equivalent to pa=a;
a[i] is equivalent to *(a+i)

difference between the array name and the pointer variable

pa=a; pa++; O.K.
a=pa; a++; p=&a; Wrong!! Forbidden!!

If addresses p q refer to the same array, than relations <, <=, > >= work properly.
Dangerous may be comparison of addresses to different arrays!

Examples:

/* Version I */
strlen(char *s) /* give the string length */
{
    char *p=s;
    while (*p != '\0')
        p++;
    return (p-s);
}

/* Version II */
strlen(char *s) /* give the string length */
{
    char *p=s;
    while (*p)
        p++;
    return (p-s);
}

/* Version III */
strlen(char *s) /* give the string length */
{
    char *p=s;
    while (*p)
        ;
    return (p-s);
}

/* Version IV */
strlen(char *s) /* give the string length */
{
    char *p=s;
    while (*p)
        ;
    return (p-s);
}

Arithmetic on addresses

Typical operations
p++; - shift to the next element
p+=i; - shift to the element placed i positions ahead of the actual

The following operations are proper pointer operations:
• assignment of pointers to objects of the same type
• addition or subtraction of pointers and integer numbers
• subtraction or comparison of two pointers to the elements of the same array
• assignment of value zero to a pointer or comparison of the pointer with zero

It is forbidden:
• neither add themselves two pointers nor multiply them, or divide shift, connect with masks, add to them floating pointer numbers (anyhow pointers are not integer numbers)
• assign without casting to a pointer to an object of type a pointer to an object of another type (except pointers to an object of type void *)

Arguments - passing by value

In C language all function arguments are passed by "value". It means that function receives copies of arguments and works on them. Function called could not directly change value of variable in the calling function.
Example:

/* Function rises argument base to power n */
int power(int base, int n) {
    int p;
    for (p=1; n > 0; n--)
        p=p*base;
    return p;
}
Change of n variable value inside the function does not influence
the value of the argument with which it was called.

Multidimensional arrays

Basic rules:

• two-dimensional array is a one-dimensional array, at which
each element is a one dimensional array.

• it is allowed not to fully determine the first dimension (its
final size determination is then carried out at the moment
of initialization), but all other dimensions should be spe-
cified.

• elements are stored in memory rowwise, i.e. recall
daytab[i][j] /* [row][column] */
denotes a reference to the i-th row,
j-th column. (Most rapidly varies the
right index.)

Example:
static char daytab[2][13]=
{0,31,28,31,30,31,31,30,31,30,31,31,30},
{0,31,28,31,30,31,31,30,31,30,31,31,30};

int day_of_year(int year, int month, int day)
{int i,leap;
leap=year % 4 == 0 && year % 100 != 0
|| year % 400 == 0;
for (i=1; i<month; i++)
day += daytab[leap];
return day;
}

Initialization of multidimensional
arrays

List of initialisers of its elements surrounded by square brac-

• If in some square bracket some elements are missed, then
they are supplemented with zeros ( {0} ).

• If in some internal brackets appear all initializers, such
brackets may be omitted.

• square brackets surrounding list of characters may be re-
placed by string composed of exactly such characters.

• omitting in an array declaration determination of the num-
ber of elements in the first array dimension causes its de-
fault determination on the basis of the initialisers list.

Examples:

int Vec[3]={10,20,30};
long int Arr[3][2]={(1,2), (3,4), (5,6)};
char Greet[6]={'H','e','l','l','o'};
char Text[]="Hello World";
short int Matrix[2][3]=
{(1,2),(3,)};

Pointers and multidimensional ar-
rays

After the following definitions:
int a[10][20];
int *b[10];
both notations a[3][4] and b[3][4] are correct references to single
objects of type int.

a is a true multidimensional array, 200 places of size int are
reserved for it; element a[row][column] one finds according
to the formula: 20*row+column

b assigns only 10 places for pointers and does not initialize
them; assignment of initial values must be done directly
- statically or by the program.

If each element of array b points to an array with 20 integer
elements, then we have reserved 200 places of size int plus
10 cells for pointers.

An important advantage of a pointers array is the possibility to
differentiate the rows lengths.

Passing two-dimensional array to function

We have to determine the number of columns. Hence, if an
array daytab should be passed to the function f, then the function
declaration have one of the three forms given below:
f(int daytab[2][13]);
f(int daytab[][13]);
f(int (*daytab)[13]);

The last declaration says, that daytab is a pointer to the array of
13 integer numbers. Parentheses are in this case necessary, since
rectangular brackets [] has got a higher priority than operator
of indirect addressing *. Without parentheses declaration
int *daytab[13];
introduces an array of 13 pointers to integer objects.

Possibility of using multidimen-
sional tables of negative indices

Solution assumed by the authors of Numerical Recipes:
#include <stdlib.h>
#include <stdio.h>
#define TYPFLOAT long double
#define MSQRT sqrtl
#define MFABS fabsl
void nrerror(char error_text[])
{
    fprintf(stderr,
"Numerical Recipes run-time error...\n\n");
    fprintf(stderr,"%s\n",error_text);
    fprintf(stderr,"...now exiting to system...\n\n");
}
exit(1);

TYPFLOAT **dmatrix(int nrl, int nrh, int ncl, int nch)
{
    int i;
    TYPFLOAT **m;

    m=(TYPFLOAT **) malloc((unsigned) (nrh-nrl+1)*sizeof(TYPFLOAT*));
    if (!m) nrerror("allocation failure 1 in dmatrix()");
    m -= nrl;

    for(i=nrl;i<=nrh;i++) {
        m[i]=(TYPFLOAT *) malloc((unsigned) (nch-ncl+1)*sizeof(TYPFLOAT));
        if (!m[i]) nrerror("allocation failure 2 in dmatrix()");
        m[i] -= ncl;
    }
    return m;
}

void free_dmatrix(TYPFLOAT **m, int nrl, int nrh, int ncl, int nch)
{
    int i;
    for(i=nrh;i>=nrl;i--) free((char*) (m[i]+ncl));
    free((char*) (m+nrl));
}

Calls to the function dmatrix may look as follows:

TYPFLOAT **Q;
/* ....... */
Q=dmatrix(-10,N,-5,N);