Data Structures and Algorithms (DSA) Course 2

Iulian Năstac

Matrices (recapitulation 1)

- Syntax:

  type arrayName [lim_1] [lim_2] … [lim_n];

- where ***lim_i*** is the limit of index ***i*** (on dimension ***i***)
- The index ***i*** can have the following values:

  0, 1, 2, … , lim_i - 1

Matrices (recapitulation 2)

Notes:

- Each lim_i (where i = 0, 1, 2, …, n) is a positive integer.
- We can access the elements of a matrix by using variables with indices.
- In C, the name of a matrix is in fact a pointer that indicates the address of its first element.
- At the declaration of a matrix, the compiler allows a memory zone to contiguous store all its elements.
- **Matrices and pointers are strong related**.

Matrices (recapitulation 3)

Vectors

- How to use a vector as a parameter to a function

Examples:

a) ***void funct (int *x)*** /* pointer*/

{.....

}

b) ***void funct (int x[10])*** /* vector with mentioned dimension */

{.....

}

c) ***void funct (int x[ ])*** /* vector with unmentioned dimension */

{.....

}

Matrices (recapitulation 4)

Strings in C

The character string is one of the most widely used applications that involves vectors.

Standard functions for strings

- calculating the length of strings (eg. strlen)
- copying the strings (eg. strcpy)
- concatenating the strings (eg. strcat)
- comparing the strings (eg. strcmp)
- searching (identification) of character or substrings (eg. strchr, strstr)

Matrices (recapitulation 5)

Two-Dimensional Arrays

type *arrayName*[lim_1][lim_2]

- Two-dimensional matrices as function parameters

```
void funct(int x[ ][10])
{
    ...
}
```

- **Matrices of strings**

Multidimensional arrays

- The C language allows the use of arrays with more than two dimensions. Maximum size could depend, sometimes, by the version of the compiler.

- The general format of a multidimensional array is:

type name[lim_1][lim_2]...[lim_n];

Example: A 4-dimensional array of characters (with the dimension 10×4×5×10)

char mat[10][4][5][10];

requires 10 x 4 x 5 x 10 = 2000 bytes.

- In addition, if the matrix uses other data type like:

- integer – then 4000 bytes are required;
- double - then 16000 bytes are required.

Comments:

- The required memory greatly increases according with the number of dimensions.
- By using the multidimensional matrices, computer takes time to process the indices, therefore the access to the matrix elements will be slow.
- When a multidimensional array is used as a function parameter, then we have to declare all sizes excepting the one from extreme left.
- The variation of indices is faster for the ones on the right side when assessing a matrix that is stored in computer memory.

For example, by considering the matrix:

int mat[4][3][6][5];

Then function that receives the above matrix as argument will be defined in the following way:

```
void funct(int d[ ][3][6][5])
    {...
    }
```

Indexing pointers

- The name of an array (without mentioning any indices) is in fact a pointer to the first element of the array.

Examples:

1) *char p[10];*
   / The following instructions are identical */*
   p; <=> ***&p[0];***
   The expression: ***p==&p[0]*** *is true;*

2) *int *p,i[10];*
   p=i;
   p[5]=100; <=> ****(p+5)=100***;
   / identical instructions that have the same result */*

3) The same goes for multidimensional arrays.
   int a[10][10];
   a <=> ***&a[0][0];***
   a[1][2] <=> ****(a+12);***

It highlights the following set of rules :

- For a two-dimensional array :

type a[lim1][lim2];

The element: **a[i][k]** <=> ***(a + i·lim2 + k)**

- For a three-dimensional matrix :

type a[lim1][lim2][lim3];

The element: **a[i1][i2][i3]** <=> ***(a+i1·lim2·lim3+i2·lim3+i3)**

- The generalization for a matrix with an arbitrary number of dimensions is now obvious.

Comments:

- The pointers are often used to access the elements of an array since the arithmetic of the pointers is faster than the classic access (with indices) of the matrix elements.

- A two-dimensional array can be reduced to a pointer to a one-dimensional array (see/remember matrix of strings).

Example: A function that displays a specified row of a matrix:

```
int num[10][10]; /* a global variable */
...
void display_row(int j)
{ int *p,t;
  p=&num[j][0]; /* takes the address of the first element of the row j*/
  for (t=0; t<10; t++) printf("%d",*(p+t));
}
```

Observation: ***&num[j][0]*** is equivalent with ***num[j]***

We improve the previous example by setting the followings parameters: the row, its length, and a pointer to the first element in the array

```
void display_row(int j, int row_length, int *p)
{  int t;
    p=p+(j* row_length);
   for(t=0;t< row_length; t++)
        printf("%d",*(p+t));
}
```

Notes:

- In C, a three-dimensional matrix can be reduced to a pointer to a two-dimensional array, which can be further reduced to a pointer to a one-dimensional array.

- By generalization, a **n-**dimensional matrix size can be reduced to a pointer to a **n-1** dimensional array. And so on, the process continues, until it comes to an one-dimensional array.

For example, by having the matrix :

int mat[lim_1][lim_2][lim_3]...[lim_n];

then the element ***mat[i_1][i_2]...[i_n]*** is equivalent to:

****(mat + i_1·lim_2·lim_3·...·lim_n +***
i_2·lim_3·lim_4·...·lim_n + ... + i_(n-1)·lim_n +
i_n)

Array initialization

Any matrix can be directly initialized starting with its declaration:

type matrix_name[lim1][lim2]...[limn]={list of values};

Note: the more to the right is an index, then it varies much faster, when considering the appropriate matrix elements of computer memory (this is a key issue which concerns access by indicators).

Examples:

1) In the statement:

int i[10]={1,2,3,4,5,6,7,8,9,10};

i[0] has the value **1**;

i[9] has the value **10**.

2) Arrays of strings allow short initialization :

char matrix_name[dimension]= ”string of characters”;

3) The statement:

char str[14]=”C programming”;

is similar to:

char str[14]={’C’,’ ’, ‘P’,’r’,’o’,’g’,’r’,’a’,’m’,’m’,’i’,’n’,’g’,’\0’};

As a remark, do not omit the addition of '\0' to the end of a string, when it is not used the classic initialization for strings (with quotes).

4) A two-dimensional array initialization:

```
int matrix[3][2]   =        { 1, 1,
                              2, 4,
                              3, 5
                            };
```

5) It is possible to initialize arrays without providing theirs size:

char er[ ]=”This message is stored in the computer memory \n”;

6) Dimensionless matrix initialization is not strictly restricted to the one-dimensional arrays. But for the multidimensional matrices, one must specify all dimensions except the leftmost.

For example:

```
int mat[ ][2]   =      { 1, 1,
                         2, 4,
                         3, 5,
                       };
```

The advantage is that it can increase or shorten the table without changing the dimensions of the matrix (referring to the first dimension of this example, which remains unspecified).

Pointers
1. Introduction

- A pointer is a variable that contains an address in the computer memory, where there is stored the value of another variable.
- Consequently, pointers are used to refer to such data which are known through their addresses.
- **Harold Lawson** is credited with the 1964 invention of the pointer, for introducing this concept into PL/I, thus providing for the first time, the capability to flexibly treat linked lists in a general-purpose high level language.

The advantages of using pointers:

- Pointers offers the possibility to change the arguments of a function.

- Pointers facilitates dynamic memory allocation.

- Pointers can improve the efficiency of certain routines.

2. Declaring pointers

The declaration of a pointer type variable shall conform to the following format :

type *name_pointer;

As long as the declaration of a ordinary variable is ***type name***, we can say that the form ***type ** from a pointer statement is in fact a new kind of type (pointer type).

Notes:

- The base type (*type* from the construction of *type**) of the pointer defines the type of the variable that can be stored in that memory which is addressed by the pointer.

- The pointers' arithmetic is created relative to theirs base type, therefore it is essential to properly declare a pointer.

3. Operators for pointers

There are different operations that can be executed with pointers, but there are two special operators which are used in unary expressions like:

operator variable

These two operators are:

- ***indirection operator*** (*) → it operates on a pointer variable, and returns an l-value equivalent to the value at the pointer address.

- ***reference operator*** (&) → acts on an lvalue and the result is a pointer.

Notes:

- Unary operators **&** and * have priority over all arithmetic operators, except unary minus that has the same order of precedence.

- We must ensure that the pointer variables used always the correct type of data.

The following program can be compiled without error but does not produce the desired result

```
#include<stdio.h>
int main()
{ double x = 100.3, y;
   int *p;
   p=&x;  /* forces p to point to a double */
   y=*p;   /* will not work as expected */
…
}
```

- In the previous example it will not be assign the value of **x** to **y** because **p** is a pointer of type integer and only two bytes of information will be transferred to y (not all 8 bytes which normally form a floating point number of type double).

- Moreover, in C ++ is prohibited converting a pointer into another one without explicit use of a cast conversion.

Note:

The symbol * has in ***C*** a number of four uses, completely different in expressions like:

- ***type *name*** – to declare a pointer.
- ****name*** – which denote an *lvalue*.
- ***op_1 * op_2*** – which expresses a multiplication.
- ***/* … */*** – for comments.

4. Expressions with pointers

4.1. Assignment instructions for pointers

A pointer can be assigned with:

- a memory address (usually obtained with &);
- another pointer.

As observation, the format specifier used in output functions (such as **printf**) to display the value of a pointer is %p.

Example:

```
#include<stdio.h>
int main(int)
{ int x=100;
   int *p1, *p2;
   p1=&x ;
   p2=p1 ;
   printf("%p", p2) ; /* display the address of x, but not
                                its value*/
   printf("%d", *p2); /* display the value of x */
}
```

4.2. The arithmetic of pointers

- Arithmetic operations that can be performed using pointers are: addition (**+**) and subtraction (**-**), and supplementary, the incrementation (**++**) and decrementation (**- -**).

- **It should be noted that the pointer arithmetic is relative to their base type.**

Examples:

1) ***int *p1;*** /* suppose p1 has a value of 2000 */
/* variables of type int needs 2 bytes */

After expression

p1++;

p1 will contain 2002 and not 2001.
And if we have the expression:

p1 - -;

and p1 with the initial value of 2000, then p1 will get 1998 after decrementation.

2) Concerning the placement of a variable in memory, we present two hypothetical cases (and we point out that they are not simultaneous) :

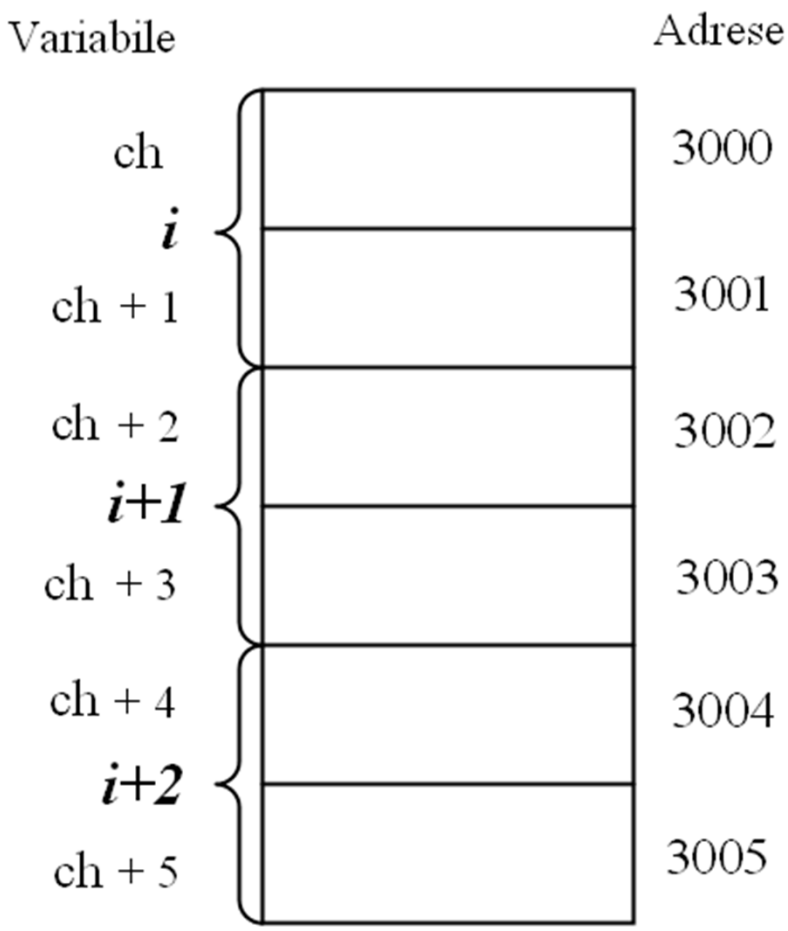
*char *ch=3000;*

and

*int *i=3000;*

!

The first pointer is incremented sequentially for five times, and the second for two times (into an equivalent space of memory). If we superimpose the two hypothetical cases, then the local memory area would look like:



3) We are not limited (for pointers) only to operations of incrementation or decrementation. We can add or subtract integers to (or from) pointers.
If p1 and p2 are pointers of the same type, then the expression:

p1=p2+12;

makes p1 to indicate the 12th element (ordered in memory) of the same type as the one that p2 indicates.

4) Considering the code sequence:

```
...
int x,y;
int *p;
...
```

we present some equivalences:

y=x+100; ⇔ *p=&x;*
*y=*p+100;*

x=y; ⇔ *p=&x;* **p=y;* ⇔ *p=&y;* *x=*p;*

x++; ⇔ *p=&x;* *(*p)++;*

Notes:

- The other arithmetic operators (excepting +, ++, -, şi - -) are prohibited in operations with pointers (or memory addresses).
- You can not add or subtract variable or constant of type float or type double from pointers.
- The arithmetic of the pointers should not be confused with the classic arithmetic used by other variables (where all operations are usually permitted, depending on the situation).

4.3. Using a pointer with several types of data

- There are cases where the same pointer can be used for multiple data types in the same program, but not simultaneously. This can be done by initially declaring the void type for the pointer:

void *name;

```
...
int x;
float y;
char c;
void *p;
...
p=&x;
...
p=&y;
...
p=&c;
...
```

The pointer variable p can assign addresses in memory areas which may contain data of different types (*int*, *float*, *char*, etc.) only if using cast conversions.

A cast conversion:

(type) operand

For the example above, the expression:

**p=10;*

is not correct, while

**(int *)p=10;*

is a correct one.

The Rule of Implicit conversion

It works when a binary operator is applied to two operands.

The steps of the rule :

- First convert the operands of type ***char*** and ***enum*** to the type ***int***;
- If the current operator is applied to operands of the same type then the result will be the same type. If the result is a value outside the limits of the type, then the result is wrong (exceedances occur).
- If the binary operator is applied to operands of different types, then a conversion is necessary, as in the following cases:

- If one operand is **long double**, therefore the other one is converted to **long double** and long double is the result type.

- Otherwise, if one operand is **double**, therefore the other one is converted to **double** and **double** is the result type.

- Otherwise, if one operand is **float**, therefore the other one is converted to **float** and **float** is the result type.

- Otherwise, if one operand is **unsigned long**, therefore the other one is converted to **unsigned long** and **unsigned long** is the result type.

- Otherwise, if one operand is **long**, therefore the other one is converted to **long** and **long** is the result type.

- Otherwise, if one operand is **unsigned**, therefore the other one is converted to **unsigned** and **unsigned** is the result type.

4.4. Comparing the pointers

- We can compare two pointers in a relational expression.

- **But… in which situation??**

Note: At different runs of the same program, the compiler can place the variables used (in a new configuration) at memory-swapped locations

When it is justified to compare the pointers?

- This comparison is however justified only if the two pointers indicate to the elements from the same array (matrix).

- Otherwise, the effect of the comparison is irrelevant as long as the compiler places the variables at different addresses depending on the available memory of the computer.

Example:

- Generate a list headed by the LIFO principle (LIFO – Last Input First Output). This is a stack that will store and provide integer values, according to the numbers entered. Thus, if you enter :
  - A value other than 0 or -1 → it is placed at the top of the stack;
  - 0 → remove a value from the stack;
  - -1 → it stops the program .

```
#include<stdio.h>
#include<stdlib.h>
#define DIMENSION 50
void push(int i);
int pop(void);
int *b, *p, stack[DIMENSION];
int main()
{       int value;
        b=stack;  /* b shows the base of the stack */
        p=stack;   /* initializes p */
        do
          {printf("\n Enter the value :");
           scanf("%d", &value);
           if(value!=0)    push(value) ;
                    else printf("\n The value from the top is %d\n",
                                                pop());
         }while(value ! = -1) ;
}
```

```
void push(int i)
{
        p++;
        if(p==(b+ DIMENSION))
          {printf("\n Stack is
                    overloaded");
           exit (1);
          }
        *p = i;
}
```

```
int pop(void)
{ if(p==b)
    {printf("\n Stack is empty" );
     exit(1) ;
     }
   p - - ;
   return *(p+1) ;
}
```