POINTERS

1. Overview

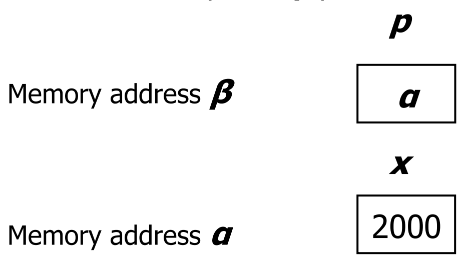
The learning objectives of this lab session are:

- To understand the C pointer types, the operations which may be performed with pointers, and how dynamic allocation/deallocation of memory is performed.
- To understand the relationship between pointers and arrays
- To learn how to access dynamically allocated arrays using pointers
- To understand and use pointers to functions

2. Brief theory reminder

2.1. The pointer type

A *pointer* is a variable whose values are addresses. If pointer ***p*** has as a value the memory address of variable ***x***, one says that ***p*** points to ***x***.



A pointer is bound to a type. If ***x*** is of type **int**, then pointer ***p*** is bound to type **int**.
Declaring a pointer is similar to declaring any variable, except for the fact that the pointer name is preceded by the character *****:

type *name;

Example:

```
int *p;
```

The address (i.e. a pointer to) a variable is obtained using the unary operator **&**, called a ***referencing operator***.

Example. Consider the declarations:

```
int x;
int *p;
```

Then, **p=&x**; has as an effect the assignment of the address of variable **x** to **p**. In the above sketch, the variable **x** is located at address **α**, and thus the value of **p** will be **α**.
To get the value stored in a memory area whose address is contained in a pointer, **p**, use the unary operator *****, called a ***dereferencing operator***.
Example:

a) Statement **x=y** is equivalent to one of the following sequences:

```
p=&x;     or   p=&y;
*p=y;          x=*p;
```

b) Statement **x++** is equivalent to the sequence:

```
p=&x;
(*p)++;
```

In these examples, **p** must be bound to the type of **x** and **y** like this:

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

There are cases when a pointer must not be bound to any data type. In this case, the following declaration is used:

```
void *name;
```

If we have the declarations:

```
int x;
float y;
void *p;
```

Then any of the statements:

```
p=&x;
p=&y;
```

is correct, but type cast must be involved, in order to specify the type of data to which **p** points:

```
(type *)p
```

Note. It is necessary to know all the time what is the type of the value stored at the address assigned to a **void *** pointer. If this is not observed and obeyed, errors can occur.

Example. Use of a pointer of type void.

```
int x;
void *p;
```

Statement **x=10** is equivalent to the sequence:

```
p=&x;
*(int *)p=10;
```

Essentially, if **p** is a pointer declared as **void *p**, one cannot use the dereferencing operator, ***p,** without specifying the type of date using a type cast.

2.2. The relationship between pointers and arrays

A name of an array has for a value the address of its first element. In consequence, one says that the name of an array is a constant pointer, which cannot be changed at run time.

Example:

```
int arr[100];
int *p;
int x;
```

```
...
p=arr;  /* p is assigned the address of the element arr[0] */
...
```

In the previous example, the assignment **x=arr[0]** is equivalent to **x=*p**;

Thus, it comes out that if an effective (actual) parameter is a single-dimensional array, then the corresponding formal parameter can be declared in two ways:

a) As an array: **type formal_parameter_name[]**;
b) As a pointer: **type *formal_parameter_name;**

The two declarations are equivalent, and the usage of the indexed variable construct :

formal_parameter_name [index];

is correct. This is illustrated in the example below, which is intended for finding the minimum and maximum element of a sequence.

```
/* Program L6Ex1.c */

/* Shows a use of an array as a formal parameter   */
#include <stdio.h>

void Max_min1(int n, int a[], int *max, int* min)
{
   int i;

   *max=a[0];
   *min=a[0];
   for (i=1; i<n; i++)
   {
      if (a[i]>*max) *max=a[i];
      else if (a[i]< *min) *min=a[i];
   }
}
void Max_min2(int n, int *a, int *max, int *min)
{
   int i;
   *max=a[0];
   *min=a[0];
   for (i=1; i<n; i++)
   {
      if (a[i]>*max) *max=a[i];
      else if (a[i]< *min) *min=a[i];
   }
}
int main()
{
   int i, n, maximum, minimum;
   int x[100];

   /* Data input */
   printf("\nMaximum and minimum element of integer array x.\nArray size is:");
   scanf("%d", &n);
   for (i=0; i<n; i++)
   {
      printf("\nx[%d]=", i);
      scanf("%d", &x[i]);
   }
   /* Call of the first procedure */
   Max_min1(n, x, &maximum, &minimum);
   printf("Max_min1: maximum=%d and minimum=%d\n", maximum, minimum);
   /* Call of the second procedure */
   Max_min2(n, x, &maximum, &minimum);
   printf("Max_min2: maximum=%d and minimum=%d\n", maximum, minimum);
   return 0;
}
```

2.3. Operations with pointers

The following operations are applicable to pointers:

a) **Increment/decrement by 1**. In this case the value of the pointer is incremented/decremented with the number of bytes needed to store a data item of the type to which the pointer is bound to. The operators used are ++ and – –.
Example:

```
int arr[100];
int *p;
     ...
p=&arr[10];
p++;   /* Value of p is incremented by sizeof(int), p has now the address of
arr[11]  */
```

b) **Addition/subtraction of an integer to/from a pointer**. The operation **p±n** has as an effect the increase (**p+n**) or the decrease (**p–n**) of the value of pointer **p** with **n*number of bytes** used to store a data item of the type to which the pointer is bound to. For the example above, if **x** is of type **int**, then
   x=arr[i];
is equivalent to::
   x=*(arr+i);

c) **The difference of two pointers**. If two pointers, **p** and **q** point to elements **i** and **j** of the same array, i.e. **p=&arr[i]** şi **q=&arr[j]**, and **j > i,** then **q-p = (j – i)*number of bytes** used to store a data item of the base-type of that array.

d) **Pointer comparison.** Two pointers which both point to the elements of the same array can be compared using the relation, and the equality operators:

   <   <= > >=   ==   !=

Below is the program previously shown, but this time rewritten to use pointer operations.

```
/* Program L6Ex2.c */

/* Shows operations with pointers */
#include <stdio.h>

void Max_min1(int n, int a[], int *max, int* min)
{
   int i;
```

```
  *max=a[0];
  *min=a[0];
  for (i=1; i<n; i++)
  {
    if (a[i] >*max) *max=a[i];
    else if (a[i] < *min) *min=a[i];
  }
}
void Max_min2(int n, int *a, int *max, int *min)
{
  int i;

  *max=*a;
  *min=*a;
  for (i=1; i<n; i++)
  {
    if (*(a+i) >*max) *max=*(a+i);
    else if (*(a+i) < *min) *min=*(a+i);
  }
}
int main()
{
  int i, n, maximum, minimum, x[100];

  /* Data input */
  printf("\nMaximum and minimum element of integer array x.\nArray size is:");
  scanf("%d", &n);
  for (i=0; i<n; i++)
  {
    printf("\nx[%d]=", i);
    scanf("%d", &x[i]);
  }
  /* Call of the first procedure */
  Max_min1(n, x, &maximum, &minimum);
  printf("Max_min1: maximum=%d and minimum=%d\n", maximum, minimum);
  /* Call of the second procedure */
  Max_min2(n, x, &maximum, &minimum);
  printf("Max_min2: maximum=%d and minimum=%d\n", maximum, minimum);
  return 0;
}
```

2.4. Dynamic allocation/deallocation of heap memory

For global and for static variables, allocation is static, i.e. variables are preserved till the program terminates.

For automatic variables, allocation is dynamic, as the stack is discarded upon return form a function.

Heap memory is a dynamic, special memory area, apart from the stack. It can be managed using specific functions, with prototypes in **stdlib.h**.

Allocation of heap memory is achieved using the functions with the prototypes:

```
void *malloc (unsigned n);
void *calloc(unsigned no_of_elem, unsigned dim);
```

malloc allocates a contiguous **n octet** area, and **calloc** allocates a contiguous of **no_of_elem * dim** octets area.

Both functions return:

- Upon **success**, a pointer to the beginning of the allocated area. (As the retrun pointer is of type void, a type cast is needed);
- Upon **failure**, the NULL pointer.

To release areas allocated with **malloc** or **calloc** the **free** function is used. Its prototype is:

void free (void *p);

C also provides a function for changing the size of a block of memory previously allocated using calloc or malloc. The prototype for **realloc** is:

void * realloc (*void *ptr, size_t newsize*)

The **realloc** function changes the size of the block whose address is *ptr* to be *newsize*.

Since the space after the end of the block may be in use, realloc may find it necessary to copy the block to a new address where more free space is available. The value of **realloc** is the new address of the block. If the block needs to be moved, realloc copies the old contents.

If you pass a **null** pointer for *ptr*, realloc behaves just like 'malloc (*newsize*)'. This can be convenient, but beware that older implementations (before ISO C) may not support this behavior, and will probably crash when realloc is passed a null pointer.

Like malloc, realloc may return a null pointer if no memory space is available to make the block bigger. When this happens, the original block is untouched; it has not been modified or relocated.

You can also use realloc to make a block smaller. The reason you would do this is to avoid tying up a lot of memory space when only a little is needed. In several allocation implementations, making a block smaller sometimes necessitates copying it, so it can fail if no other space is available.

If the new size you specify is the same as the old size, realloc is guaranteed to change nothing and return the same address that you gave.

```
/*  Program L6Ex3.c  */

#include <stdio.h>
#include <stdlib.h>

int main()
{
   char *str1,*str2;

   /* Allocate memory for the first character string */
   if ((str1 = (char *) malloc(100)) == NULL)
   {
      printf("Insufficient memory\n");
      exit(1);
   }
   printf("\nInput the first character string an press  ENTER\n");
   gets(str1);
   printf("\nThe string you supplied is\n  %s\n", str1);
   /* Allocate memory for the second character string */
   if ((str2 = (char *) calloc(100, sizeof(char))) == NULL)
   {
      printf("Insufficient memory\n");
      exit(2);
   }
   printf("\nInput the second character string an press  ENTER\n");
   gets(str2);
   printf("\nThe string you supplied is\n  %s\n", str2);
   /* Free allocated memory */
   free(str1);
   free(str2);
   return 0;
}
```

2.5. Use of a function as a parameter

The name of a function is a pointer to that function. That is why the name of a function can be used as an actual parameter in a function call.

Let **f** be a function which will be used as an actual parameter. Its header is:

type_of_f f(formal_parameter_list_for_f);

In this case, the formal parameter of a function **g** which will be invoked with **f** as an actual parameter may is presented in the header of function **g**:

type_of_g g(..., type_of_f (*p)(formal_parameter_list_for_f), ...)

An invocation is as follows::
g(..., f ,..);

Program L6Ex4.cpp shows an example. The example implements and algorithm for integrating a function using the trapezoid method. That is, the approximation formula used is:

$$\int_a^b f(x)dx = h\left(\frac{f(a)+f(b)}{2} + \sum_{i=1}^{n-1} f(a+i*h)\right)$$

where ***n*** is the number of sub-intervals in which the interval [***a***, ***b***], was divided into; the size of a sub-interval is ***h***.

```
/* Program L6Ex4.c */

/* Shows the way to use a function as a parameter */
#include <stdio.h>
#include <math.h>

double f(double x)
{
   return (3*x*x +1); /* the function to integrate; can be any function f(x) */
}

/* Computes an integral using the trapezoid approximation method */
double integral(double a, double b, int n, double(*p)(double x))
{
   int i;
   double h,s;
   h=(b-a)/n;
   s=((*p)(a)+(*p)(b))/2.0;
   for (i=1;i<n;i++)  s+(*p)(a+i*h);
   s=s*h;
   return s;
}

int main()
{
   double a,b;
   int n;
   char ch;

   /* Read the integration interval  [a,b] */
   printf("\na=");
   scanf("%lf",&a);
   printf("\nb=");
   scanf("%lf",&b);
   ch='Y';
   while (ch=='Y' || ch=='y')
   {
      printf("\nn=");
      scanf("%d",&n); /* n is the number of sub-intervals */
      printf("\nFor n=%d the value of the integral is %lf", n, integral(a, b, n, f));
      printf("\nNew value for n? [Yes=Y/y NO=any other character] ");
      ch=getchar();
   }
   return 0;
}
```

3. Lab Tasks

3.1 Execute the examples provided. Analyze the following:

- Use of an indexed variables when the name of an array was defined as a pointer in a function header (L6Ex1.cpp);
- Allowed operations with pointers. What is the advantage of replacing an indexed variable by a pointer expression? (L6Ex2.cpp);
- How is the memory area for dynamic variables allocated on the heap? (L6Ex3.cpp).
- What is the advantage of passing functions as effective parameters? (L6Ex4.cpp).

Write programs for solving the following problems:

3.2. Using only pointers and pointer expressions, write functions to read, display, and multiply a two matrices.

3.3. Using only pointers and pointer expressions, write a function to sort a vector with real number elements.

3.4. Using only pointers and pointer expressions, write a function to merge two vectors. The given vectors contain real number elements, in ascending order. The result vector must contain only the distinct elements of the two given vectors, also in ascending order.

3.5. Write a function to sort ascending ***n*** character strings***.***

3.6. Write a function which will find the root of a function, ***f***(***x***), in the interval [***a***, ***b***], knowing that this function has a single root in this interval. The function ***f*** will be passed as an effective parameter.

3.7. Write a program to draw the representation of a function, ***y***=***f***(***x***), on the screen. The function, ***f***, will be supplied in the following format:

```
double f(double x)
{
    return ....../* function to draw */
}
```

The program will contain a function which draws on the screen; this function will have the ends of the interval, the step (or the number of points) and the function to represent as formal parameters.

3.8. Write a function to compute the value of the derivative of a polynomial, ***P***(***x***), of degree ***n***, in a given point ***x=x*0**. The degree, and the coefficients of the polynomial, and the point, ***x*0**, will be passed as parameters. You must use pointers.

3.9. Write a function to compute the product of two polynomials, using only pointers and pointer expressions.

3.10. Write a function to calculate the ***k***th power of a square matrix, using pointers to access to the elements of the matrix. The resulted matrix will be displayed in natural form.

3.11. Write a function to calculate the transposed of a matrix, using pointers and pointer expressions to access to the elements of the matrix. The resulted matrices will be displayed in natural form.