Pointers and Arrays

Overview

- Pointers
- Pointers as function argument
- Arrays
- Strings

Pointers

- Pointer is a variable which contains an address
- Declare using type and asterisk

```
<type>* p_name;
```

- Get value using dereferencing

```
int* p;
…
(*p) = 10;
```

3

Pointers

The term pointers makes most C programmers shiver. Pointers are what made C for a lot of people difficult to learn. It is very useful but can be dangerous as well. Most bugs in C programs are because of the pointers.

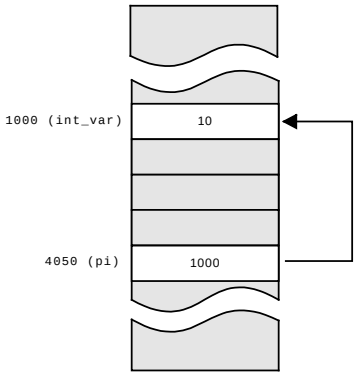
A pointer is as the name suggests a pointer to something. It points to some value. The pointer for example can point to an integer number or float number or even to another pointer. This is accomplished by storing the address of what it points to. The pointer is a variable as well but it does not store a normal value directly but it stores the address of the value thus it is pointing to the value. The declaration of a pointer has the following generic form:

*<type * var_name;*

Some examples:

```
int* pi;          /* Pointer to an integer value */
float* pf;        /* Pointer to a float value */
char* s;          /* Pointer to a character */
```

Like normal variables the pointer variable has a value of which it is initially unknown where it points to. Therefore perhaps the most important rule when using pointers is that the pointer needs to be initialised with an address. Initialisation of a pointer is a bit more difficult than that of a normal variable. It is not possible to assign just any address to a pointer. The compiler must calculate an address for us to which the pointer must point. In a simple example we can first create a piece of memory containing an int value (by declaring a variable of type int) and assign the address of this variable to a pointer.




```
int int_var = 10;
int* pi = &int_var;


/* pi contains the address of int_var, it points to int_var */
```

The '&' operator calculates the address of the variable *int_var* and assigns it to the pointer *pi*. We let the compiler assign the address of the variable because we do not know the address of the variable when creating the source code.

Although *pi* points to the variable *int_var* there is no connection between both. The pointer does not know anything of the variable even it is pointing to the same memory address. The pointer has the same address stored in it as where the variable *int_var* resides. Therefore when we change the value of *int_var* the address content is changed. The pointer still points to the same changed value.

```
int_var=30; /* Memory contents where int_var is stored is changed to 30 */
```

We know that *pi* points to an int value so if we want to change the value where it points to we have to use its address. The pointer contains the address and by using a technique called dereferencing we can get the contents of the memory it points to (the same as that of variable *int_var*).

```
(*pi)=40;  /* Contents of the address pi holds is changed to 40 */
           /* The variable int_var is therefor changed as well
           because it is stored at the address pi holds */
```

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

int main()
{
   int a = 10;
   int b = 20;
   int* p;

   /* Before using the pointer */
   printf("a = %d\n", a);            /* Will print a = 10 */
   printf("b = %d\n", b);            /* Will print b = 20 */

   p = &a;                           /* Let p point to the variable a */
   printf("p points to %d",(*p));    /* Will print 10 */
   (*p) = 30;

   p = &b;                           /* Let p point to the variable b */
   printf("p points to %d",(*p));    /* Will print 20 */
   (*p) = 40;

   /* After using the pointer */
   printf("a = %d\n", a);            /* Will print a = 30 */
   printf("b = %d\n", b);            /* Will print b = 40 */

   exit(0);
}
```

Pointers as Function Arguments

- Normally function arguments are copied on the stack
- If you don't want a copy use pointers

```
sqr(int* a);
```

- Pass argument address upon function call

```
int var = 10;
sqr(&var);
```

4

Pointers as function-arguments

Normally functions use call by value for their arguments. This means that all values passed to a function are copied onto the stack for that function. Inside the functions the variables are copies of the original supplied to that function. Changing these copies will not affect the originals. The copies are destroyed when the function ends. See example below.

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

void print_number(int nr);
void sqr(int nr);

void print_number(int nr)
{
   printf("%d\n", nr);
}

void sqr(int nr)
{
   nr = nr * nr;
}

int main()
{
   int a = 10;

   print_number(a);     /* Prints 10 */
   sqr(a);
   print_number(a);     /* prints 10 */

   exit(0);
}
```

The example shows a call to *print_number()* with variable *a* as its argument. The contents of *a* is copied to the stack and is destroyed after *print_number()* ends. Changing this value inside the function *print_number()* will only affect the copy and not the original. The same holds for the *sqr()* function. It changes the copy and not the original therefore the *sqr()* function calculates the square but the result is lost.

Using pointers we can change the function *sqr()* so that it will change the original value and not the copy. The argument passed to the function will be the address of the variable. The argument type therefore has to be changed to a variable which can store an address the pointer.

```
void sqr(int* nr);
```

The test program is changed to:

```
void print_number(int nr);
void sqr(int* nr);

void print_number(int nr)
{
   printf("%d\n", nr);
}

void sqr(int* nr)
{
   (*nr) = (*nr) * (*nr);
}

int main()
{
   int a = 10;

   print_number(a);     /* Prints 10 */
   sqr(&a);
   print_number(a);     /* prints 100 */

   exit(0);
}
```

Arrays

- An array is a list of values of one kind
- General form

```
<type> name [<size>];
```

- Use indexing value to get element of array

```
int array_int[10]; /* 10 int values */
array_int[0] = 20; /* Set element to 20 */
```

- Array starts indexing with 0 (zero)

5

Arrays

An array is a collection of values of the same type. When we declare an array in C we get a list of variables which can be referenced using the same name (the array name) with an index value specifying the variable inside that list. We create such an array by using the index operator [] with the size of the array.
The general form is:

<type> name [<size>];

The <type> can be replaced by any type possible in C so even pointers can be used. The size however has to be a constant. It cannot be a variable. Since the size cannot be changed, they are also called static arrays. Some examples:

```
int array_int[10];      /* Array of 10 integers */
float array_f[3];       /* Array of 3 float values */
```

After an array is created we can access its elements by using the index operator [] specifying which element to access.

```
array_int[0] = 10;
array_f[2] = 30.0;
```

The indexing starts with zero (0). The problem however arises when we access elements outside the boundaries of the array. For example element 100 in the *array_int*. This will not cause a compiler error. And depending on the platform, it might not even give a run-time error. The compiler does not check the indexing value neither does the system during the run-time. The effect of accessing outside the boundaries of the array results in accessing memory that is part of other variables or even memory that is not part of our program. Therefor we have to watch out that we do not index outside the boundaries.

Some rules for arrays.

- Size has to be static.
- Indexing starts with 0.
- Always a contiguous part in the memory.
- We cannot determine the size of an array. Remember the size used when creating array.

The following example shows some uses of arrays.

```
/* Program to show the use of one-dimensional arrays. */
/* (C) Datasim BV 1995 */

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

main()
{

    double my_arr[10];
    int j;

   /* Declare an array of doubles; indexing starts at 0!
      Indexing with arrays in some other languages starts at
      1 (so, be careful) */


   /* Now initialise the array to 0.0 */
   for (j = 0; j < 10; j++) my_arr[j] = 0.0;

   /* Accessing the elements; note that we loop outside the
      valid range without getting a compiler error !!!!
      This will probably result in a run-time error and
      you might will need to REBOOT your system. This is certainly
      the case on DOS machines where there is little protection
      against this type of violation!!!! */
   for (j = 0; j < 15; j++)
   {
      printf("Index: %d , Value: %f\n", j, my_arr[j]);
   }
}
```

Initialising Arrays

- Arrays need to be initialised element by element
- Cannot assign two arrays

```
array_int1 = array_int2; /* NOT POSSIBLE */
```

- Can initialise upon creation by specifying each element

```
int array_int1[4] = {1, 4, 5, 6};
float array_f[] = {10.23, 34.33, 30.0};
```

6

Initialising Arrays

Using arrays is not the same as using normal variables of type int, char, float or double. Arrays cannot be assigned. If we want to copy an array to another array we have to loop over all the elements and copy each element of that array.

```
void main()
{
    int array_1[3];
    int array_2[3];
    int counter = 0;

    /* Initialise array_1 */
    array_1[0]=10; array_1[1]=20; array_1[2]=30;

    /* Initialise array_2 (copy of array_1) */
    for(counter = 0; counter < 3; counter++)
    {
        array_2[counter] = array_1[counter];
    }
}
```

Initialising the array at creation is easier. It is possible to specify the values of each element in the array between {} and separating each value with a comma.

```
void main()
{
    int array_1[3] = {3, 4, 5};
    int array_2[] = {5, 8, 2, 4};
    int array_3[5] = {2, 7, 8};
}
```

Normally the size given is the same as the number of elements in the list. If the size is omitted (array_2) then the size will be determined from the number of elements in the list. If the number of elements in the list is smaller than the given size (array_3) the remaining elements will be initialised to zero.

Strings

- A string is an array of characters
- Declare using array notation

```
char string[80];
/* A string of 80 characters */
```

- Last character of string should be `'\0'`
- Use functions declared in `<string.h>`

7

Strings

The language C has no type for representing a string like you have in other programming languages such as Basic. To use strings we have to declare an array of characters.

```
char string_1[80]; /* An string of 80 characters */
```

The disadvantage of using an array for representing strings is that we cannot determine the size of the string during run-time. This is solved by the C language to state that the last character of a string should be the '\0' character. The following small example will fill the string and print it on the screen:

```
void main()
{
    char string[80];

    string[0] = 'T';
    string[1] = 'e';
    string[2] = 'x';
    string[3] = 't';
    string[4] = '\0';   /* Set end of string */

    printf("The string is : %s", string);
}
```

If we comply with the standard form of a string that the last character is a '\0' we can use the library `<string.h>` which contains a lot of functions for using string such as copying, comparing etc.

Using Strings

- Use functions in `<string.h>`
- Can initialise upon creation

```
char str[80] = "C the language";
```

- Use functions in `<string.h>` e.g.:

```
strcpy(str, "Some other text.");
```

8

Using Strings

We cannot use string variables as normal variables. It is not possible to assign two strings to each other. This is because the strings are actually arrays of characters and arrays cannot be assigned. If we want to copy two arrays we have to copy each element. Initialising a string can be a tough job if we initialise character by character (as we can see in the previous example).

As we saw in 'Initialising Arrays' we can specify the elements of the array upon creation. When it is an array of characters we can even initialise it with a constant string.

```
void main()
{
    char s1[] = {'C', ' ', 'l', 'a', 'n', 'g', 'u', 'a', 'g', 'e', '\0'};
    char s2[] = "C language";  /* Adds the '\0' for us */
    char s3[80];

    s3 = "Hello"; /* NOT POSSIBLE! We have to use strcpy() */
                                                strcpy(s3, "Hello");
}
```

Arrays and Pointers

- Array name is the address of the first element of the array

```
int array[3];
/* array is address of first element
   array == &array[0]*/
```

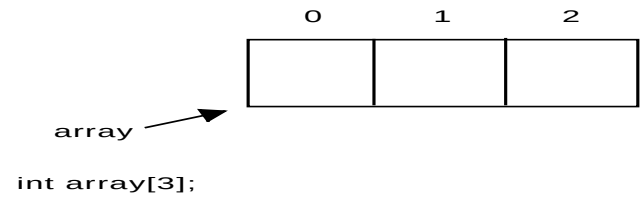
- Pointer also contains address of some element
- Can assign pointers to arrays
- Can use array indexing with pointers

9

Arrays and Pointers

An array is implemented as one contiguous piece of memory. So if element zero of an integer array is located at address 1000 than the next element is situated at address 1002. This is always guaranteed.

The array is implemented as a pointer pointing to the first element in the array.



Therefore we can also use pointers and let them point to the beginning of the array. They often say that arrays and pointers are the same. We can use the index operators for pointers:

```
void main()
{
    int array[3];
    int* p;

    p = array;   /* p points to the first element of the array */
    (*p) = 10;   /* First element is 10 */
    p[0] = 10;   /* First element is 10 */
}
```

If we use the technique of indexing with pointers we have to be sure that the pointer points to an array and not to a single value. Otherwise we are accessing memory which is not assigned to us.

Pointer Arithmetic

- It is possible to add a value to the pointer

```
int* pi;
pi = array;
pi[1] = 10;
(*(pi+1)) = 10;
```

- Adding to a pointer means adding the array type's size to the address

```
pi+1; /* Add one int size to the address stored in pi (
          next element) */
```

10

Pointer Arithmetic

It is possible with pointer variables to add a value to it. This results in adding the size of the element type to the pointer. So if it is a pointer to an *int* value, adding one will result in the addition of one *int* size (two bytes in DOS, four bytes in 32 bit Windows). If the pointers points to a *float* value this results in adding four bytes (the size of a *float*).

```
void main()
{
    int array[3];
    float farray[3];
    int* pi;
    float* pf;

    pi = array;
    pf = farray;

    (*pi+1) = 10;       /* Same as pi[1] */
    (*pf+1) = 20.0;     /* Same as pf[1] */
}
```

When using this technique we have to be sure that the pointer points to a memory block that is large enough. It is easy to add 5 to the pointer *pi* but then we access a piece of memory which is not assigned to our program. This can result in memory problems. In the previous example the pointer points to an array that is large enough so no problems here.

Multidimensional Arrays

- Can create multidimensional arrays using multiple [] pairs

```
int marray[2][3];
```

- Is an array of arrays
- The pointer equivalent is a double pointer (pointer to a pointer)

```
int** pmarr;
```

11

Multidimensional Arrays

We can create multidimensional arrays using multiple [] pairs.

```
/* Program to show the use of multi-dimensional arrays.
   (C) Datasim BV 1992 */

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

void main()
{
    double my_arr[10][5];      /* The two dimensional array */
    int i, j;                  /* Two counters */

    /* Declare a 2d array of doubles; indexing starts at 0!
       Indexing with arrays in some other languages starts at
       1 (so, be careful)
       This array can be visualised as an array of arrays. */


    /* Now initialise the array */
    for (i = 0; i < 10; i++)
    {
        for (j = 0; j < 5; j++) my_arr[i][j] = (double) (i * j);
    }

    // Access the elements
    for (i = 0; i < 10; i++)
    {
        printf("\nRow: %d : ", i);
        for (j = 0; j < 5; j++)
        {
            printf("%f ", my_arr[i][j]);
        }
    }
}
```