Pointers in C

Introduction

The purpose of this document is to provide an intuitive overview of the notion of pointers in the C programming language, along with a collection of applications that shows possible uses of pointers.

Memory and Addresses

The memory of every computer is organized as a collection of *memory locations*. Each memory location is capable of storing 1 byte of information (8 bits). One of the important properties of memory organization is the fact that each memory location has a unique *name*, that distinguishes it from all other locations. Any reference to a memory location has to be performed through its name. The name of a memory location is what we typically call the *address* of such memory location. Typically the address of a memory location is simply an integer number. This is illustrated in Figure 1.

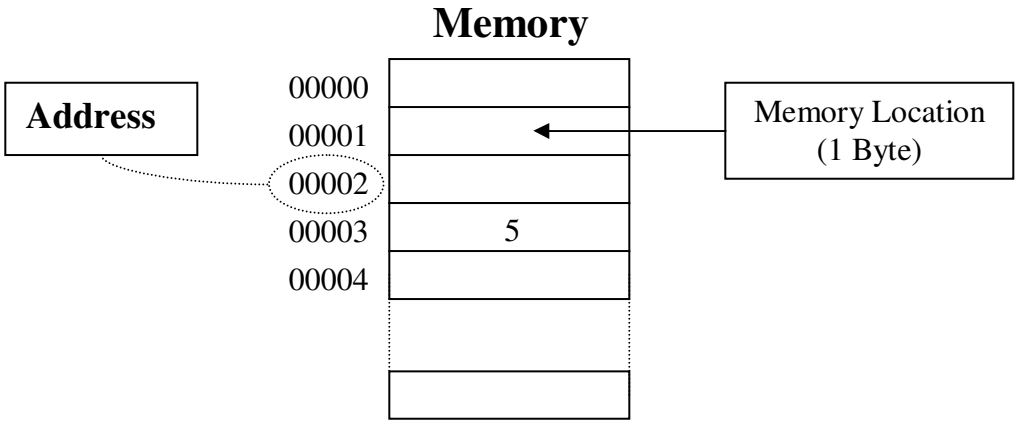

Figure 1: Memory

Observe that there is an important distinction between the content of one or more memory locations and their addresses. The address is the name that identifies one particular location, while the content of the location is the pattern of bits currently stored in that location. For example, in Figure 1, the location with address 00003 has content 5 (i.e., 5 is the pattern of bits stored in the memory location whose name is 00003).

Pointer Variables

Let us now think about what happens when we declare a variable in a C program. For example let us consider the declarations:

```
int x;
double y;
```

Each time we declare a variable in a program, we request C to reserve a certain amount of memory – i.e., identify a set of memory locations – that are going to be used to store the value of such variable. So, when we process the declaration int x, the C program will proceed to reserve a number of contiguous memory locations to store the value of the variable x. The number of locations that are going to be reserved depends on the type of the variable. C provides a function, called `sizeof`, that takes as input the name of a type and returns as result the number of memory locations that an object of such type will require.
In the previous example, the program will proceed in reserving sizeof(int) locations for the variable x and sizeof(double) locations for the variable y (see Figure 2).

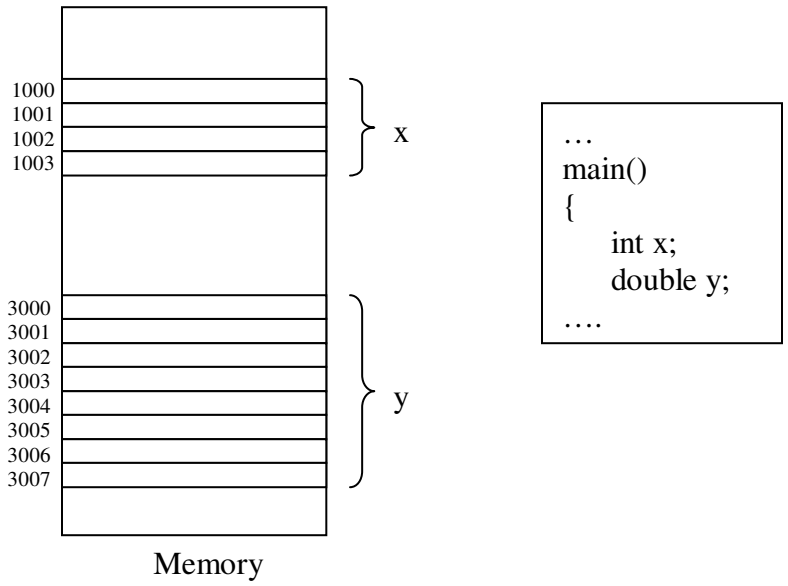

Figure 2: Creation of two variables

Each variable created in a program will occupy a number of contiguous locations in memory, starting at a particular address. In the example in Figure 2, we have that the variable x has been created starting at the location wit address 1000 (and it occupies locations 1000, 1001, 1002, and 1003), while the variable y has been created starting at the location with address 3000. With a little abuse of terminology, we will start talking about the *Address of a Variable*; the address of a variable is the address of the first location occupied by the variable. So, in the example in Figure 2, the address of the variable x is 1000, while the address of y is 3000.
It is clear that each variable that is created in a program has an address.

Declarations

We are now ready to start talking about pointers. There is nothing mysterious about pointers – in C, a pointer is simply a special type of variable. Just as any other variable in a program, a pointer has to be declared, it will have a value, a scope, a lifetime, a name, and it will occupy a certain number of memory locations. So what is special about pointers? The only peculiar aspect is that the value of a pointer variable is always a *memory address*. In simple words, a pointer is a variable that is used to store addresses.

They are called pointers for the simple reason that, by storing an address, they "point" to a particular point in memory. In particular, we will be interested in using pointers to keep track of where other variables are in the memory of the system (i.e., we would like the pointers to "point" to other variables).
In C there is an additional restriction on pointers: they are *not* allowed to store *any* memory address, but they can only store addresses of variables of a given type. So we will have pointers that will store addresses of integer variables, pointers that will store the addresses of double variables, pointers that will store the addresses of char variables, etc. There is a reason behind this restriction: by restricting what kind of addresses we can store in a pointer variable, we will be able to safely know what kind of objects we will find by jumping at the specified memory address.
So let us start being a bit more practical. We would like to declare a variable of type pointer, called p, and we would like to use this variable to keep track of where some integer objects are located in the memory. Thus, p will always contain addresses of integer objects. The pointer can be created with the declaration:

```
int *p;
```

In this declaration, p is the variable name, and int * is the type (look at it as a single name). The * indicates that you are declaring a pointer – the variable will always contain an address. The int will indicate that you are interested in storing addresses of objects of type integer. Similarly, let us assume that we have a structure used to represent points in the 2-d space

```
struct point
{
     double x;
     double y;
};
```

If we want to have a variable q that will be used to store the address of an object of type struct point, then its declaration is

```
struct point * q;
```

You can have pointers to any type of object. Of course, in the moment you declare p to be a pointer to integer, you will not be allowed to use it to store addresses of other objects of other types (e.g., you will not be allowed to store in p the address of an object of type char or of type struct point).

Once again, let me stress that a pointer is just a variable; the value of such variable is always an address; it will be the address of some other object that you have created in your program (e.g., the address of some other variable).

Operations on Pointers

The next natural question is: how can we compute the address of an object? Without such facility pointers would be useless (yes, you can store an address in a pointer, but where

do these addresses come from???). C provides a single operation, denoted by the simple &, which allows you to compute the address of an existing object (e.g., the address of an existing variable). The & operation can be applied to any existing variable. The result is going to be the address of such variable (i.e., it tells you where this variable is located in the memory). This operation is called "address of".
Let us look back at the example in Figure 2; if we perform &x we get as result the address 1000, since this is the position of x in memory. Similarly, if we compute &y we get the address 3000 as result.
What do we do with the result of the & operation? Typically we take the resulting address and store it in a pointer variable. Since a pointer variable can store an address, then the result of the & operation can be legally assigned to a pointer.

Thus if we have the following pointers and the following variables declared:

```
int x;
double y;
int *p;
double *q;
```

then legal operations that we can perform are

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

The first assignment will determine the address of the variable x and store such address in the pointer variable p; similarly, the second will compute the address of y and store it in the pointer q. Now we have that p can be used to determine where x is in memory while q is telling us where is y in the memory.
Note that the operation:

```
p = &y;
```

is illegal. p is a pointer, but it can only contain the address of integer objects; &y gives us the address of a double object, so we cannot store such address in p.

Figure 3 is a picture that shows the effect of the previous assignments. Observe that in the figure, the variable p contains the address 1000 – i.e., it is pointing to the object that is stored at address 1000 (i.e., the variable x).
It is also important to observe the effect of assigning one pointer to another one. Consider the following declaration:

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

Now, let us assume that we have performed the assignments

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

At this point in time we have that p is a pointer variable whose value is the address of x (i.e., p tells us where is x in the memory), while q is pointing to the variable y. Let us now perform the assignment

p = q;

As in any other assignment, the effect is to copy the value of the variable q into the variable p. Thus, after the assignment p and q are two variables with the same value. Since the value of q was the address of y, then after the assignment p contains the address of y as well. Thus, p and q are both pointing to y. This is illustrated in Figure 4.

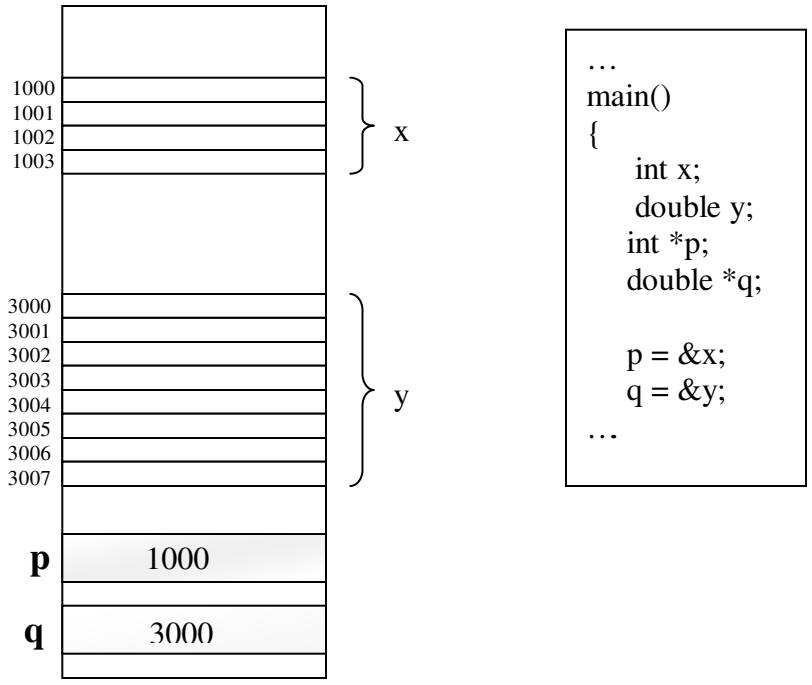

Figure 3: Execution of Assignments to pointers

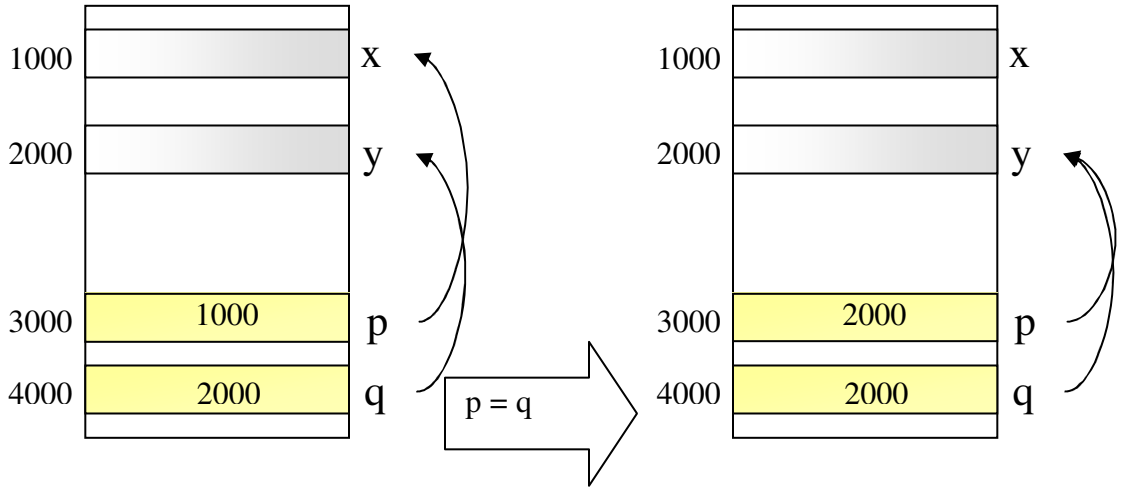

Figure 4: Execution of p=q assignment between pointers

C provides a second operation that can be used with pointers. The operation is called *dereference* and it is denoted by the symbol *. The operation * can be *only applied to pointers*. The effect of applying the dereference operation to a pointer is to access the memory indicated by the address stored in the pointer. Effectively, the * operation can be seen as the opposite of the & operation. While the & operation computes the address of an object, the * operation takes an address and gives you back the object located at such address. It is VERY IMPORTANT to remember that the * operation can be applied ONLY to pointers.
For example: let us assume we have

```
int x;
int y;
int *p;
int *q;

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

If we apply the * operation to the pointer p (this is written as *p), we obtain as result the object which is located at the address stored in p – now, since p contains the pointer to the object x, by doing *p we obtain the object located at such address, which is simply x. If we continue with the previous example, if we write:

```
*p = 12;
```

we store 12 in the object *p, i.e., the object that is located at the address stored in p; but since p contains the address of x, then this is the same as

```
x = 12;
```

Similarly, if we write

```
*q = *p – 2;
```

we are doing the following: getting the object *p – i.e., the object which is located at the address stored in p – and subtract 2 from it. Since p contains the address of x, then *p is x itself, so the right hand side of the assignment is the same as x-2. The left hand side is the object located at the address in q; since q contains the address of y, then *q is y itself. So the assignment is the same as y = x – 2.

Note that if are in the situation illustrated in the Figure 4 (on the right), then *p, *q, and y are denoting the same object.

One last observation: let x be a variable; what is * (& x) ? Given x, the operation &x will give us the address of x; now we take this address and dereference it – i.e., you get the object that is located at such address; but the object located at address &x is x! Thus *(&x) is simply x.

Pointers Arithmetic

Pointer variables are variables that are used to store addresses. This means that if we have a pointer variable, e.g.,

```
double *q;
```

then the value of q at any point in time is an address (specifically, the address of some double object). An address is really just an integer number. Thus, it may seem reasonable

to ask whether arithmetic operations can be performed on variable of type pointers. E.g., can we write things like

```
q++;
```

The answer is positive, although only certain operations are allowed. The operations which are allowed on pointers are the following:

- it is allowed to add an integer number to a pointer
- it is allowed to subtract an integer from a pointer
- it is allowed to subtract a pointer from another pointer.

Furthermore, it is allowed to compare pointers using the comparisons:

- == to check if two pointers have the same value
- != to check if two pointers have a different value
- >, <, >=, and <=

Now let us consider the following example. Let us assume that we have

```
double *p;
double x;

p = &x;
```

Thus p is a pointer that contains the address of the variable x. Let us assume that the variable x is located in memory at the address 1024. Now the question is: what is going to be the value of p after the instruction

```
p++;
```

is executed?

Arithmetic operations on pointers have a behavior that is different from what one may expect. The result of p++ is not 1025! Since p is pointing to an object of type double, then by doing p++ what we are asking is to obtain the address of the *next object of type double in memory*. Since each object of type double occupies more than one location – to be precise each object of type double occupies sizeof(double) locations – then by doing p++ we obtain the address 1024+sizeof(double). Let us assume, for example, that sizeof(double) is 8, then by doing p++ we obtain that p contains 1024+8 = 1032.

Thus, when we have a pointer p, and we perform an operation of the type p+n, the result will be the address of p increase by n*sizeof(base type), where the base type is the type of the objects p is pointing to.

As another example,

```
struct point *r;
```

r is a pointer to a struct point object. Let us assume that r contains the address 2000. Let us also assume that sizeof(struct point) is 22. Now, if we perform the operation r = r+3; then the result will be that r contains the address 2000+22*3 = 2066.

The subtraction operation performs in the same way. If we have a pointer r then by doing r-5 we obtain an address equal to the address in r minus 5 times the size of the object r is pointing to. If r is a pointer to a double that contains 2000, and each double takes 8 bytes, then r-5 will be the address 2000-5*8=1960.

Now it is important to observe one thing: if we perform an operation like p++ on a pointer p, we access the area of memory that immediately follows *p (i.e., we go to the "next" object). But in general, if p is pointing to some variable (e.g., x), then by doing p++ we really don't know what we are going to access. A sequence of operations like the following one

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

p = &x;
p++;
```

is very dangerous. Initially p is pointing to the object x (i.e., p contains the address of x). After the operation p++ is performed, p is pointing to the object that is immediately after x in memory; but C does not guarantee anything about what is going to be present in the location that follow x in memory. So if we proceed and perform the operation

```
*p = 20;
```

we may produce a serious error – we are trying to store 20 in the area pointed by p, which probably does not even belong to the program.

The only situation where pointer arithmetic makes sense is when we are using pointers to point to elements of an array: in that case we know that there are many variables (the elements of the array) that have been created in contiguous locations in memory; thus moving from one object to the next one will be ok.

For example, consider the following declarations:

```
double array[10];
double *p;
```

The array is composed by 10 double variables, which are created one after the other in memory. Let us assume that the array is created starting at location 2000 and let us assume that each double occupies 8 bytes.

If we perform the operation

```
p = &array[0];
```

then we are storing in p the address of the object array[0]. Since the array starts at address 2000 and array[0] is the first element of the array, then p will contain the address 2000.

Now let us assume we perform the operation

```
p++;
```

then what we are doing is changing the pointer p so that it points to the "next double" in memory; the content of p after this operation is 2008. But observe that 2008 is simply the address of the object array[1] – i.e., we moved from one element of the array to the next one. Similarly, if we perform next

```
p = p + 3;
```

then we change the address by adding to it 3 times the size of a double; since initially p is 2008, then we get that p contains 2008+24 = 2032. Before the operation p was a pointer to the element array[1], now we jumped three elements ahead, so p will be a pointer to the element array[4].

In the same way, if we perform next the operation

```
p = p – 2;
```

it means that we want to move the pointer back two positions (i.e., move two doubles back), so if p was initially 2032, then after the operation p will contain 2032-2*8 = 2016. When we started p was a pointer to the element array[4], now we move two positions back and we will have a pointer to the element array[2].

Thus when we deal with arrays, we can perform pointer arithmetic to move a pointer back and forth inside the array, moving from element to element inside the array.

As an example, let us assume that we have an array of integers and we would like to add together all the elements of the array. The typical structure of the code is

```
int array[SIZE];
int sum = 0;

for (i=0; i < SIZE; i++)
        sum = sum + array[i];
```

Now we can rewrite the same code without the need of using the array notation, but using pointers instead; the code is as follows:

```
int array[SIZE];
int sum = 0;
int *p;

// initially we set p to point to the beginning of the array
p = &array[0];

for (i=0; i < SIZE; i++)
{
        sum = sum + (*p);
        p++;
}
```

observe that in this case we are using a pointer to access the different elements of the array; We start by having a pointer p that points to the first element of the array. By doing sum = sum+(*p) we take the object (*p), which is simply array[0], and add it to the sum. After that, the pointer p is incremented – so that it points to the next element of the array (thus p is a pointer to array[1]). At the next iteration, we perform again sum = sum +(*p), but now *p is the object pointed by p which is array[1]. And so on.
The same code can be also written as:

```
int array[SIZE];
int sum = 0;
int *p;

p = &array[0];
for (i=0; i < SIZE; i++)
        sum = sum + *(p + i);
```

In this case, at each iteration we are adding to the sum the object *(p+i). Since p is always pointing to the first element of the array, then p+i is a pointer to the element that is i positions from the first one – but that's simply the pointer to array[i].

There is a very deep relationship between pointers and arrays. Indeed an array is really nothing else than a pointer to an area of memory. When we write the declaration

```
int x[10];
```
what really happens is that an areay of memory big enough to store 10 integers is created, and a pointer to such area of memory is stored in x. Thus, x is really a pointer that points always to the beginning of the array. We can explicitly use x as a pointer if we want. Thus, if we have
```
int x[10];
int *p;
```
and we want to assign to p the address of the first element of the array, we can do this in two different ways: either
```
p = &x[0];
```
or we can simply write
```
p = x;
```
This also means that if we want to use pointer arithmetic to access the elements of an array (as we have done in the previous example), we can simply write
```
int array[SIZE];
int sum = 0;

for (i=0; i < SIZE; i++)
        sum = sum + *(array + i);
```
Note that we use the name of the array simply as a pointer and then we use pointer arithmetic (array+i) to determine the address of the different elements of the array.
The relationship between arrays and pointers goes even further: each time we write
```
array[i]
```
in our program, what we really intend is to access the element of index i in the array; this element has address array+i; thus the notation array[i] is equivalent to
```
*(array + i)
```
Indeed each time the compiler finds array[i] in the code it simply removes it and replaces it with *(array+i).
We can take advantage of this fact. Consider the following example: we have an array of 100 integers
```
int array[100];
```
and we would like to repeatedly access the last 10 elements of such array. For example we need to add together these 10 elements:
```
sum = 0;
for (i=90; i < 100; i++)
        sum = sum + array[i];
```
This may be a bit confusing (having loops with strange starting points). To make our life simpler we can look at the last 10 elements of the array as a separate array (of size 10). This can be simply accomplished by setting a pointer to this smaller array:
```
int *smallarray;
…
smallarray = &array[90];
```
at this point I can forget how I have obtained smallarray and treat it simply as the name of an array; thus to add the elements together I can just write:
```
sum = 0;
for (i=0; i < 10; i++)
```

```
sum = sum + sumarray[i];
```

Finally, let us make some considerations concerning the use of pointers in connection with structures. Structures are just another type of variables, thus we can compute also addresses of structures and store them in pointer variables. Let us consider the following example; we have a structure

```
struct point
{
        double x;
        double y;
};
```

Given this structure, we can declare variable of type struct point:

```
main ()
{
        struct point a, b;
```

At this point we can declare variables that can store addresses of objects of type struct point:

```
        struct point *ptr;
```

The value of such pointer can be computed, for example, as follows

```
        ptr = &a;
```

Now ptr becomes a pointer to the object a; remember that a is a structure in memory (containing two double variables), while ptr is a variable that currently contains the address of a (i.e., ptr is telling me where a is located in memory).

We can now access the components of the structure by following the pointer ptr; for example, if we want to store 2.5 in the y component of the structure pointed by ptr we can write

```
        (*ptr).y = 2.5
```

note the following:

- we want to perform three operations: follow the pointer, that will lead us to a structure; access one component of the structure (y); assign a special value to such component.
- the following of the pointer has to be done first; for this reason we have enclosed *ptr between parenthesis, to ensure that it is done first;
- once we have reached the structure, now we need to select what component to use; the traditional notation '.y' is used for this purpose;
- finally we can assign the value to the component.

Note that we need to use parenthesis; in the precedence of the operations in C, we have that the '.' operation is actually stronger than '*', thus if we write

```
        *ptr.y = 2.5
```

what actually happens is that we first apply . between ptr and y and then we apply * to the result. The compiler will generate an error since ptr is not a structure (thus we cannot apply '.' to it).

Since it is easy to confuse this, C provides an alternative notation that can be used *exclusively* when we deal with pointer to structures. The expression (*ptr).y can be rewritten as

```
        p->y = 2.5;
```

In the expression p->y, p has to be a **pointer to a structure**, while y has to be the name of one of the components of the structure. The effect of p->y is to follow the pointer p, which will lead us to a structure, and then select the component y of that structure.

Applications of Pointers

Pointers to Implement Call by Reference

The first typical application of pointers is to support call by reference. Let us start by reminding what call by reference is.

When we make a function call, we typically want to communicate some arguments to the function. C makes use of only one mechanism to communicate arguments to a function: call by value. Call by value means that when we call a function, a copy of the values of the arguments is created an given to the function. For example:

```
…
main()
{
        int x,y;
        …
        x = 5;
        y = 10;
        swap (x,y);
        …
}

void swap (int a, int b)
{
        int temp;

        temp = a;
        a = b;
        b = temp;
}
```

Now when the function swap is called, the system automatically creates two new variables (in this case called a and b) and these will contain a copy of the values that are specified in the function call (i.e., the value of x and the value of y). All the operations performed by the function operates on these copies of the values (a, b), and will not affect the original values (x, y). This is show in Figure 5.

Of course, in this particular example the function will probably not accomplish what we really want. The function swap is used to exchange the content of two variables, but when we make the call the function will receive and operate on copies of the variables, leaving the original variables (x, y) untouched. So when the function is finished, the effect of the changes done by swap are lost (the copies created when the function is called are destroyed when the function is completed).

That's a common situation in C. Each function always receives copies of values and the function does not have any way of modifying the value of variables that exist outside of the function (e.g., the x, y in the example).
So how do we allow a function to modify a variable that exists outside of the function? This is what call by reference does! Call by reference means that when we call the function we do not create copies of values but we allow the function to access the original values. This also means that if the function modifies such values, then the modification will affect the original value and will persist once the function is finished.
In C call by reference does not exist, but it can be simulated through the use of pointers. If I want a function to be able to modify a certain variable, what I can do is to give to the function information about where the variable is in memory (i.e., its address); if the function knows where the variable is in memory, by using pointers it will be able to access that area of memory and change its content.
Consider the following simple example: we want to allow a function foo to set an integer variable to zero. To make this possible, foo will need to receive an information that specifies where the variable that has to be set to zero is located in memory. This will arrive to the function in the form of a pointer. The function will follow the pointer, access the variable and set it to zero:

```
void foo ( int  * p )
{
        *p = 0;
}
```

this will accomplish an effect similar to call by reference, since the function will access an object (*p) that exists outside of the function, and change its value. The call to this function will look as follows:

```
…
int x;
…
foo ( &x );
```

note that when we call the function we need to provide an address – the address of the variable that we want the function to modify; in this case we want the function to change the value of the variable x, thus we give to foo the address of x.

If we follow this idea, we can make the swap function work correctly; the idea is that the input to swap should be now a pair of addresses, specifically the addresses of the two variables whose content should be swapped:

```
void swap (int *a, int *b)
{
        int temp;

        temp = *a;
        *a = *b;
        *b = temp;
}
```

When the function is called, we will need to provide the addresses of the two variables to be swapped:

```
main()
{
        int x, y;
        …
        swap ( &x, &y);
        …
}
```

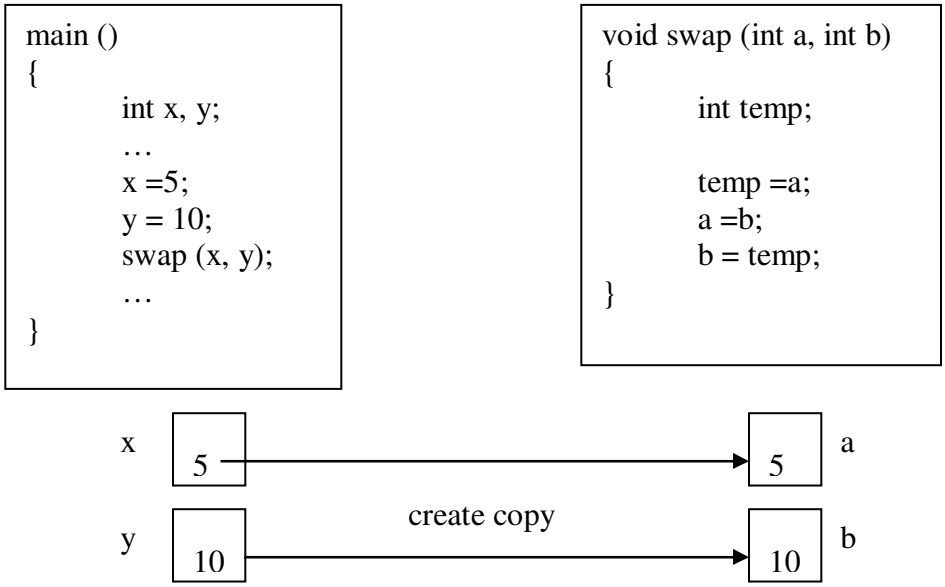

Figure 5: Call by Value

A typical situation where call by reference is useful is when we want to have a function that returns more than one result. For example, let us assume that we want to have a function that takes as input an array of characters (and its size) and returns as results the character that occurs more frequently along with the frequency of such character. Thus the function needs to return two values as result. When we use the return statement to send back a result, we can specify only one value, not two. The way to get around the problem is to use call by reference. This means that the function will receive some additional arguments, containing pointers; these pointers are telling to the function where the different results need to be stored once the execution is completed. The following is the function:

```
void count (char x[], int size, char *mostfreq, int *freq)
{
        int i;
        int counters[256] = {0};
```

```
        for (i=0; i < size; i++)
                counters[x[i]]++;

        max = 0;
        for (i=0; i < 256; i++)
        {
                if (counters[i] > counters[max])
                        max = i;
        }
        // now time to return the two results
        *mostfreq = max;
        *freq = counters[max];
}
```

When the function is called, we will need to provide the pointers to the variables where the most frequent character and its frequency will be stored:

```
main()
{
        char array[SIZE];
        int frequency;
        char character;
        …
        count( array , SIZE , &character, &frequency );
        …
}
```

In the function call we are telling to the function that the address &character is used to identify the place in memory where the most frequently occurring character should be deposited; similar thing for the frequency. After the function completes, the variable character will contain the most frequently occurring character and the variable frequency will contain its frequency.

Pointers for Dynamic Memory Management

The second important use of pointers is to support dynamic memory management.
Up to this point, if we need to have one object created, we will have to declare a variable for that object. This means that we need to know exactly when we write a program how many variables are needed. This is often infeasible. For example, if we need to have an array that contains some data, if the array is declared as a variable we need to fix its size immediately at the time the program is written. What if the size of the array will be known only when the program is executed? Ideally we would like to request the creation of the array during the execution of the program (that's what we call a "dynamic" creation of the array).
Pointers make this possible. C provides a function (called **malloc**) that, whenever called, will create a brand new object. The function can be called as many times as we want (e.g., it can be placed inside a loop) and each time will lead to the creation of a new object. The only things that this function needs are:

- it needs to know the size of the object that we want to create
- it needs to know where to store the information about this new object (so that it can be used later in the program)

The first piece of information is simply a number: it tells how many bytes of memory are needed by the object. In most of the cases, we will make use of the sizeof function to determine how much memory is needed. Once the object is created, the function will return the address of the new object – this address will have to be stored in a pointer variable. Note that this pointer is the only way to access this object that has just been created.

Let's see some examples; let us assume we want to create one variable of type double, dynamically: the code will have

```
double *p;              // to be used to keep the address of the new object
…
p = malloc (sizeof (double) );
```

The malloc takes as input the number of bytes required and gives us back the pointer to the new object. If we want to store 2.5 in the new object just created we will simply use the pointer to access it:

```
*p = 2.5;
```

As another example, let us assume that we have a structure called foo, declared as follows:

```
struct foo
{
        double a;
        int b;
        char c;
};
```

and let us assume that during the execution of the program we want to create on the fly one of such structures:

```
main ()
{
        …
        struct foo *p;  // pointer to keep track of the new object when created
        …
        p = malloc ( sizeof ( struct foo ) );
        …
```

If now we want to store 2.15 in the component called a of this object and the character 'z' in the component called c, then we write:

```
        (*p).a = 2.15;
        (*p).c = 'z';
```

One of the typical uses of malloc is to generate arrays dynamically. Consider the following situation: we have a program where we want to read a bunch of characters from the user and store them into an array; the problem is that when we write the program we do not know exactly how many characters the user will type. On the other hand, when we run the program, we are allowed to ask the user for the number of

characters. This will allow us to create the array dynamically and then use it to store the characters. The program is the following:

```
#include <stdio.h>
#include “simpio.h”

main()
{
        int size;       // will tell us how many characters are there
        char *p;        // will tell us where is the array created dynamically
        int i;

        printf(“How many characters? “);
        size = GetInteger();

        // create the array
        p = malloc (sizeof(char) * size);

        for (i=0; i < size; i++)
                *(p+i) = getchar();
}
```

The array is created simply as a chunk of memory big enough to contain size characters. We access the elements of the array simply by using the pointer p (which indicates the beginning of the array) and using pointer arithmetic to move inside the array (note that p+i is the address of the element of index i of the array).
Using the considerations we have done earlier regarding the equivalence between arrays and pointers, I can also rewrite the code more simply as:

```
#include <stdio.h>
#include “simpio.h”

main()
{
        int size;       // will tell us how many characters are there
        char *p;        // will tell us where is the array created dynamically
        int i;

        printf(“How many characters? “);
        size = GetInteger();

        // create the array
        p = malloc (sizeof(char) * size);

        for (i=0; i < size; i++)
                p[i] = getchar();
        }
```

This is possible because each time we write p[i], the compiler will automatically replace it with *(p+i). Thus effectively, after we have created the array with malloc, we can as well forget how the array was created and just use the pointer as the name of the array.

Some considerations:

- the malloc operation may fail if not enough memory is available. The malloc function will tell us that there is a failure by returning the NULL pointer as result. Thus it is always a good idea to check the result of the malloc before using it:

```
p = malloc( …. );
if (p == NULL)
{
        printf("Malloc failed\n");
        exit(0);
}
```

- there is a function that has the opposite effect of the malloc – it is used to destroy areas of memory previously created with malloc. The function is called **free** and it takes a single argument – a pointer to the area of memory that we would like to destroy. Once destroyed, the area of memory is no longer available and it should not be used any more.

Let us see another example that involves the dynamic creation of an array. We would like to write a function that takes an array of characters as input, and it creates a new array containing the same elements as the input array but in the reverse order. In order to solve this problem we need to allow the function to create the new array explicitly using a malloc. Observe that a solution that does not use malloc would be incorrect. For example, let us consider the following solution:

```
char *reverse (char x[], int size)
{
        int i;
        char new[100];

        for (i=0; i < size; i++)
                new[i] = x[size-i-1];
        return (new);
}
```

Note that the function returns a pointer to a character, which represents the beginning of the resulting array. But this function is **wrong**. The problem here is that the array new is a local variable of the function reverse, and when the function completes this array is destroyed. But the result returned with the 'return(new)' statement is a pointer to the area of memory that contains the array new. Thus the function returns a pointer to an area of memory that is immediately after destroyed. Accessing the area of memory of new from outside the reverse function would lead to bad errors.

The problem can be resolved by making use of malloc to create the array new. The interesting thing is that objects created with malloc have a lifetime that is completely controlled by the programmer, which means that the programmer requests the creation of

the object (using malloc) and he is the only one that can request its destruction (using free). So even if the array is created with malloc inside the function reverse, it will still exist when reverse finishes. The correct implementation of reverse is the following:

```
char * reverse (char x[], int size)
{
        int i;
        char *new;      // for the dynamic creation of new array

        new = malloc (sizeof(char) * size);
        if (new == NULL)
        {
                printf("malloc failed \n");
                exit(0);
        }

        for (i=0; i < size; i++)
                new[i] = x[size – i – 1];

        return (new);
}
```

Let us see another example that involves the use of pointers, in this case in connection with structures.
We would like to have a structure that represents a point in the 2-d space. A good definition of such structure is

```
struct point
{
        double x;
        double y;
};
```

The problem at hand is to allow the user to insert the coordinates of a bunch of points and store all of them in memory. We assume that we do not have any knowledge about how many points will be inserted. A first approach would be to simply ask the user how many points he/she wants and then create an array of the appropriate size. This would proceed as follows:

```
main( )
{
        struct point *array;    // this will be used for the dynamically created array
        int size;               // this will tell us later on the size of the array
        int i;

        printf("How many points do you want? ");
```

```
        scanf("%d", &size);

        // create the array
        array = malloc (sizeof(struct point) * size);
        if (array == NULL)
        {
                printf("Unable to cre ate array\n");
                exit(0);
        }

        // read the points, one by one, and place them in the array
        for (i=0; i<size; i++)
        {
                printf("x and y coordinates of the point: ");
                scanf("%d", &array[i].x);
                scanf("%d", &array[i].y);
        }
}
```

This solution is still not satisfactory, since the user needs to know in advance how many points are going to be there. It would be nicer to allow to create one point at the time and stop when the user is done. For example, let us assume that the user keeps inserting coordinates until the user types the coordinate (0,0) (which means end of input).
In this case, what we would like to create one point at the time, instead of allocating a priori a big chunk of memory to store all elements. Thus, each time we want a new point, we simply allocate the memory for that particular point

```
        struct point *new;
        …
        new = malloc (sizeof(struct point));
```

The malloc operation will be repeated each time we have a new point. Now, the problem is that the points will be spread in the memory and we do not have a way of keeping track of where they are and in what order they are meant to be accessed.
The intuition is to allow each point to keep track of its position in the sequence of points, by maintaining an additional piece of information with the point; this extra piece of information is aimed at suggesting what point is going to follow the current one in the sequence of points. The type of data organization is intuitively represented in Figure 6.
As we can see from the figure, we have an explicit information that identifies the object in memory that contains the first point created. Then each point contains an extra piece of information used to identify what point is following in the sequence. Clearly, if each point is a structure created in memory, then the connection between structures (i.e., the arrows in Figure 6) can be encoded as pointers. Thus, each point will store an address that identifies the position in memory of the structure containing the "next" point.

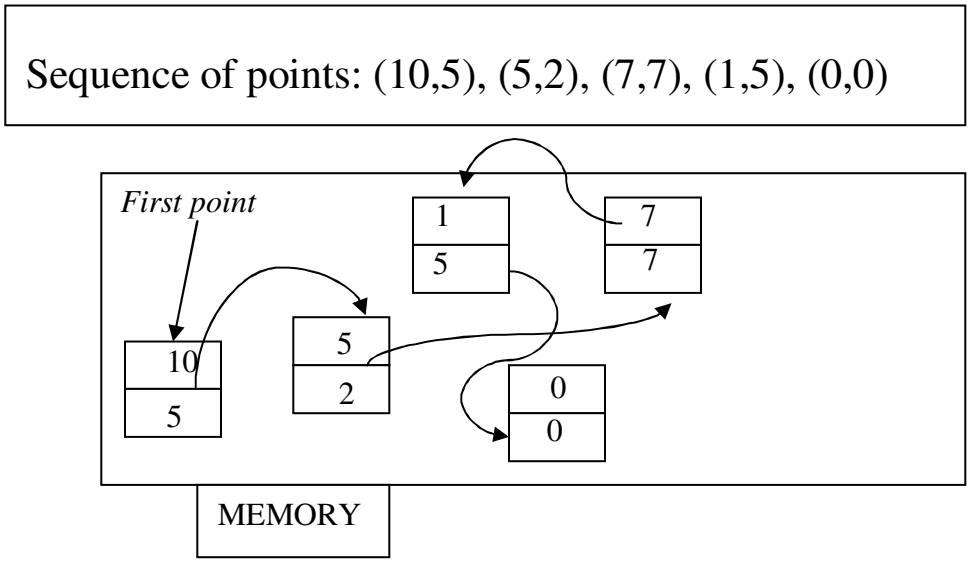

Figure 6: Points created separately and connected via links

Let us now see how all this can be encoded in C. First of all, we need to modify the definition of the structure to allow the pointer to be stored; the new definition is

```
struct point
{
        double x;
        double y;
        struct point *next;
};
```

Note that the new component (called next) is of type pointer (*), and in particular it is going to point to another struct point.
The algorithm to read the points and store them will proceed as follows

```
main()
{
        struct point *first;       // this will keep track of the first point of the sequence
        struct point *old, *new;         // used during the creation of the points

        // create the first point
        first = malloc (sizeof(struct point));
        if (first == NULL)
        {
                printf("Cannot creat e point\n");
                exit(0);
        }
```

```
        printf("Insert the coordinates of the first point: ");
        scanf("%f", &(first ->x));
        scanf("%f", &(first ->y);
        // note: first points to a structure; first->x accesses the component x

        // set the pointer old to the current node
        old = first;

        while (old->x != 0 || old->y != 0)       // repeat until both coordinates are 0
        {
                // create a new point
                new = malloc (sizeof (struct point));
                if (new == NULL)
                {
                        printf("Cannot create point \n");
                        exit(0);
                }

                // read coordinates
                printf("Insert coordinates of point: ");
                scanf("%f", &(new ->x));
                scanf("%f", &(new ->y));

                // now connect the previous point to the new one
                old->next = new;

                // move the old pointer to the new node
                old = new;
        }
}
```

Note that in the body of the loop, we are using two pointers: the pointer old refers to the point created in the previous iteration, while the pointer new refers to the new point just created. The statement old->next = new allows us to connect one point to the other in the correct order.