POINTERS IN C

University of Windsor

$\boxed{\text{1- Introduction}}$

Pointers are crucial to C, they are used in particular to:

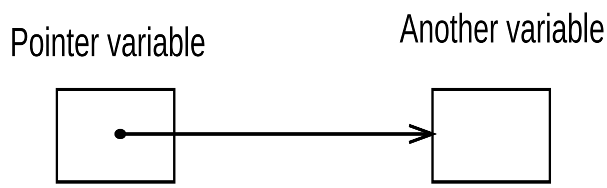
- provide the means by which functions can modify their calling arguments,
- support the dynamic allocation of memory,
- refer to a large data structure in a compact way,
- support data structures such as linked lists.

Most C programs use heavily pointers.
$\Longrightarrow$ An effective C programmer has to understand how pointers work in C.

C language is rather a middle level language!

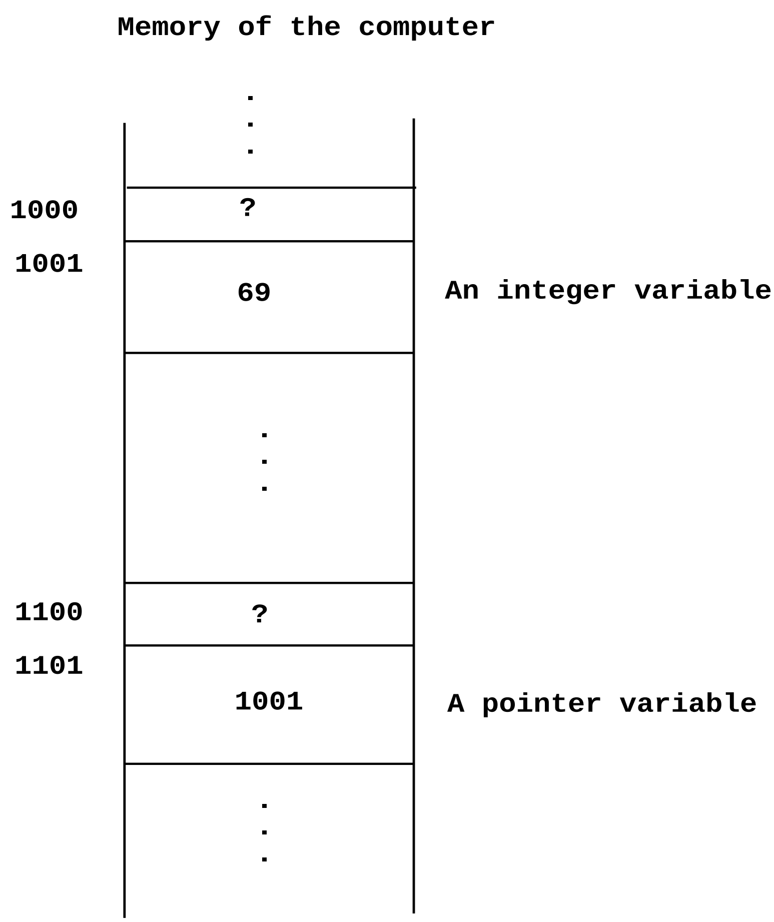
2- Pointers are addresses

A pointer is a variable whose values are memory addresses.



A pointer variable is similar to other variables: it requires memory and stores a value. However, the purpose of a pointer is special.

- A data variable $\longrightarrow$ stores data (e.g., grade).
- A pointer variable $\longrightarrow$ stores memory addresses.



Syntax: *base_type *pointer_name;*
where *base_type* defines the type of variable the pointer can point to.

Example : *int *ptr;*
ptr is a pointer-to-integer variable.

3- Manipulation of pointers

- Operators :

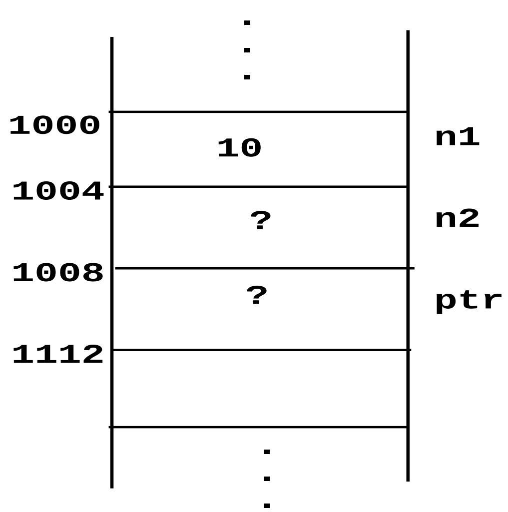
  & : a unary operator that returns the address of its operand.

  $*$: a unary operator used to **dereference** its operand, a pointer. To **dereference** the pointer **ptr** is to use the expression $*$**ptr** which is the cell whose address is stored in **ptr**.

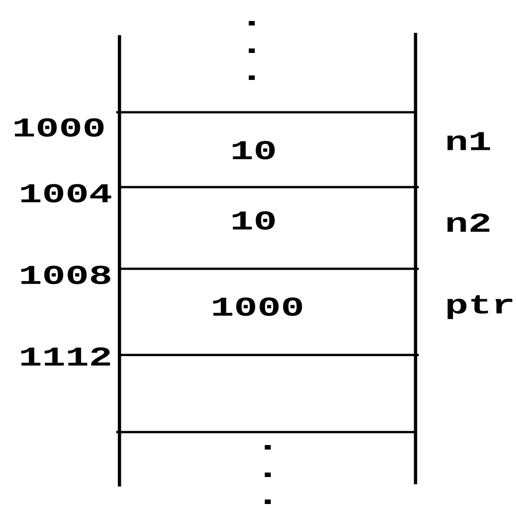
- Pointers in assignments : pointers can be used in assignments as other variables.

Example : 01

```
main(){
  int n1=10, n2;            /* 2 variables of type integer */
  int *ptr;                 /* a pointer to integer */


  ptr = &n1;             /* ptr <-- address of n1 */

  n2 = *ptr;           /* n2 <-- value stored at address ptr(n1) */

  printf(‘‘%d  %d  %d’’, n1, n2, *ptr);  /* On the screen:         */

  return(0);
}
```



Decalaration of
n1, n2 and ptr



The contents of n1, n2,
and ptr before the end.

Example : 02

```
Assume the declarations:
         float z;
         int x, y;
         int *ptr1 = &x,  /* Like any variable, pointers can be
             *ptr2 = &y;     initialized at declaration time */


The syntax of the following assignments is correct:
         ptr2 = ptr1;    /* ptr2 receives the contents of ptr1 which
                            is the address of x in that case */

         *ptr1 = *ptr2;  /* this is equivalent to x=y  */
         *ptr2 = 4;      /* this is equivalent to y=4  */
          x  = *ptr2;    /* this is equivalent to x=y  */


However, the syntax of the following assignments is not correct:
         ptr1 = x;      /* type mismatch: assigning a data
                           value (integer) to a pointer */

         y = ptr1;      /* type mismatch: assigning an address
                           to an integer variable */
         ptr1=&z;       /* this is wrong because ptr1 is supposed
                           to point to an integer variable */
```

Important : be very careful when pointers are involved in assignments. In particular, if the left side of an assignment is an address variable (pointer) then the right side must be an address.

4- The special pointer NULL

We need sometimes to indicate that a pointer is not pointing to any valid data. For that purpose, the constant *NULL* is defined in *stdlib.h* and can be used in a C program.

A common use of *NULL* : When a function returning a pointer wants to indicate a failure operation, the function will return *NULL*.

Example :

```
int *afunction();       /* a function that returns a pointer
                            to an integer */
int *ptr;                /* a pointer to integer */
      .
      .
      .
ptr = afunction();     /* ptr <-- an address to an integer */

if (ptr == NULL)        /* afunction() was unsuccessful */
   take-appropriate-action
```

Important : Do not dereference pointers that are not initialized or whose value is *NULL*.

5- Pointer arithmetic : + and -

When a pointer is incremented (resp. decremented), it will point to the memory location of the next (resp. previous) element of its base type.

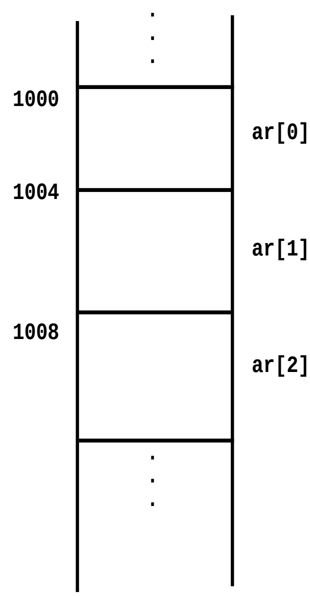
Example : Assume that *ptr*, a pointer to int, contains the address 200 and n an integer. Assume also, that the size of an integer is 4 bytes.

- ptr++ $\Longrightarrow$ ptr will contain 200 + 1×4=204.
- ptr-- $\Longrightarrow$ ptr will contain 200 - 1×4=196.
- ptr = ptr + n $\Longrightarrow$ ptr will contain 200 + n×4.
- ptr = ptr - n $\Longrightarrow$ ptr will contain 200 - n×4.

6- Pointers and arrays

Pointers and arrays are closely related.
In particular, the name of an array is the address of the array's first element.

For example, the declaration $int\ \ ar[3]$ reserves 3 consecutive units of memory. Each unit has the same size, size of an integer (e.g., 4 bytes).
In memory, this array looks like :



In particular,
$\&ar[0] = 1000$, $\&ar[1] = 1004$, $\&ar[2] = 1008$.
In general, $\&ar[i] = ar + i \times 4$(size of integer).
ar corresponds to the address 1000.
$\Longrightarrow$ ar is a pointer except that it is a constant.

Pointers and arrays are interchangeable!

Consider the example :

```
main(){
   int ar[4] = {5, 10, 15, 20};
   int *ptr;
   int i;

   for (i=0; i<4; i++)
      printf(``%d ``, ar[i]);

 /* on the screen will appear                 */

   ptr=ar;                    /* Equivalent to ptr=&ar[0] */
  for (i=0; i<4; i++)
      printf(``%d ``, ptr[i]);

 /* on the screen will appear                 */



}
```

Important : Because the name of an array is a constant pointer, it is illegal to do ar = ptr.

This program will do exactly the same job as the previous one.

```
main(){
   int ar[4] = {5, 10, 15, 20};
   int *ptr;
   int i;

   ptr=ar;    /* Equivalent to ptr=&ar[0] */

   for (i=0; i<4; i++)
      printf(``%d ``, *(ptr + i));
   /* on the screen will appear                */


  for (i=0; i<4; i++)
      printf(``%d ``, *(ar + i));
   /* on the screen will appear                 */
}
```

In other words, for C language

$$\mathrm{ptr} + \mathrm{i} \Leftrightarrow \&\mathrm{ptr}[\mathrm{i}]$$

As a consequence

$$*(\mathrm{ptr} + \mathrm{i}) \Leftrightarrow \mathrm{ptr}[\mathrm{i}]$$

Note : C "assumes" all the time that a pointer is pointing to an array!

$\Longrightarrow$ The programmer is responsible for the interpretation.

Example :

```
main(){
    int n=10;
    int *ptr=&n;      /* ptr initialized to address of n  */



    printf(``%d  %d  %d\n'', n, *ptr, ptr[0]);
    /* On the screen:                     */

    ptr[0] = 20;    /* Equivalent to *ptr = 20 */
    ptr[1] = 30;    /* Equivalent to *(ptr + 1) = 20 */
                    /* ERROR, not detected at compilation time */

   ptr++;           /* ptr contains (address of n) + 4 */
   *ptr = 30;      /* ERROR, not detected at compilation time */

}
```

Limitations on pointers arithmetic

- $*$, $/$, and $\%$ cannot be used with pointers.
- $+$ and - are restricted : only integer offsets can be added to or subtracted from a pointer.
- Two pointers cannot be added together
- A pointer p1 can be subtracted from another pointer p2. The result is an integer, the number of elements between p1 and p2.

  Example :

```
main(){
      int ar[100];
      int *p1, *p2;

      p1=ar;
      p2=&ar[60];

      printf(``%d\n'', p2 - p1);  /* On the screen:    */
}
```

  In other words, if p1 and p2 are pointers to integer containing for instance 300 and 400 respectively, p2 - p1 will be equal to $(400 - 300)/4 = 25$ assuming 4 for the size of an integer.

7- The type pointer-to-void

A pointer variable ptr defined as **void** $*$ptr is a generic pointer variable. That is, a pointer that can point to any type.

Advantage:
a pointer to void may be converted, without a cast operation, to a pointer to another data type.

Important : 'void $*$' means the type *pointer-to-void* and should not be confused with **void** which is used to indicate that a function return no value or empty parameter list.

Example :

```
void  *myfunction(); /* a function that returns a generic pointer */
int    n1;
int   *ptr1;
float  n2;
float *ptr2;


ptr2 = &n1;            /* Error */

ptr1 = &n1;            /* OK */
ptr2 = &n2;            /* OK */

ptr1 = myfunction();   /* OK 'void *' is casted to 'int *'  */

ptr2 = myfunction();   /* OK, 'void *' is casted to 'float *'  */
```

Important : Pointers to void cannot be dereferenced

Example :

```
int  n;
void *ptr=&n;

*ptr = 25;  /* Error, the type of *ptr is unknown */
```

8- Dynamic memory allocation

There are 3 styles of memory allocation in C :

- Static allocation : a variable's memory is allocated and persists throughout the entire life of the program. This is the case of *global variables*.

- Automatic allocation : When *local variables* are declared inside a function, the space for these variables is allocated when the function is called (starts) and is freed when the function terminates. This is also the case of *parameter variables*.

- Dynamic allocation : Allow a program, at the execution time, to allocate memory when needed and to free it when it is no longer needed.
  Advantage : it is often impossible to know, prior to the execution time, the size of memory needed. For example, the size of an array.

Through its standard library (include stdlib.h), C provides functions for allocating new memory from the **heap** (available unused storage). The most commonly used functions for managing dynamic memory are:

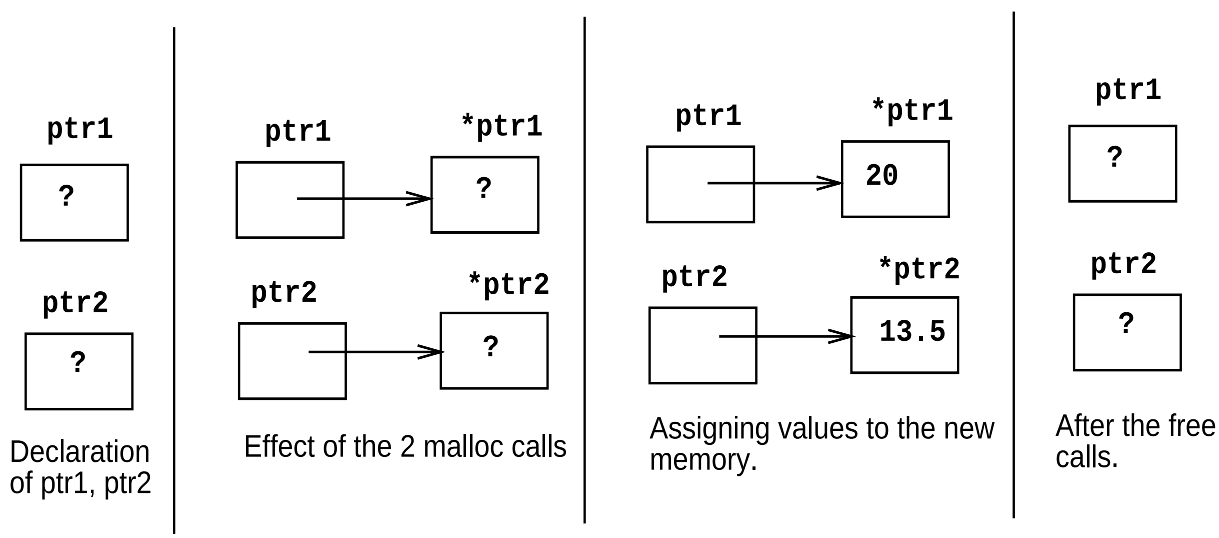
- *void ∗malloc(int size)*: to allocate a block (number of bytes) of memory of a given size and returns a pointer to this newly allocated block.

- *void free(void *ptr)*: to free a previously allocated block of memory.

Note: *sizeof* is an operator often used with *malloc*. It returns the size in bytes of its operand(a data type name or a variable name). For instance, sizeof(char)=1

Example :

```
main(){
       int *ptr1;
       float *ptr2;

      /* allocate space for an integer */
       ptr1 = malloc (sizeof(int));

      /* allocate space for a float */
       ptr2 = malloc (sizeof(float));

       *ptr1 = 20;
       *ptr2 =13.5;

        free(ptr1);   /* free previously allocated space */
        free(ptr2);
 }
```



Dynamic arrays

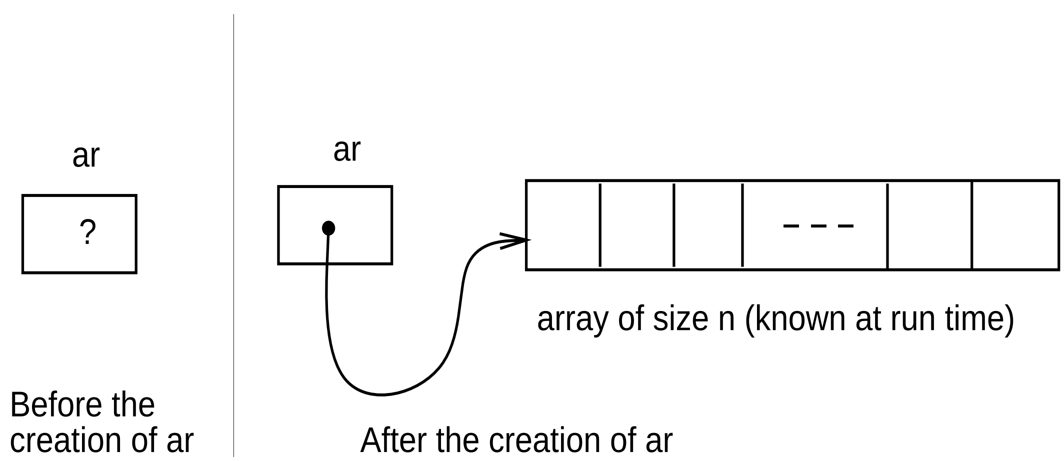
When the size of an array is not known before the execution time, allocating arrays dynamically is a good solution.

The steps for creating a dynamic array are:

1. Declare a pointer with an appropriate base type.
2. Call *malloc* to allocate memory for the elements of the array. The argument of malloc is equal to the desired size of the array multiplied by the size in bytes of each element of the array.
3. Assign the result of *malloc* to the pointer variable.

Example :

```
main(){
      float *ar;    /* Declare a pointer to float variable */
      int i, n;

      printf(``enter size of list > ``);
      scanf(``%d'', &n);        /* read the desired size */

      ar = malloc(n * sizeof(float)); /* create the array */

      for(i=0; i<n; i++)        /* ar is an array of size n */
         scanf(``%f'', &ar[i]);
      .
      .
      .
      free(ar);
 }
```



Declared arrays vs. dynamic arrays

- The name of a declared array is a constant pointer and cannot be changed whereas, the name of a dynamic array can be changed and assigned a different memory address.
- The memory associated with a declared array is allocated automatically when the function containing the declaration is called whereas, the memory associated with a dynamic array is not allocated until *malloc* is called.
- The size of a declared array is a constant and must be known prior to the execution time whereas, the size of a dynamic array can be of any size and need not to be known in advance.

Important:

If the heap runs out of space, *malloc* will fail to allocate memory and will return the pointer *NULL*. It is therefore, the responsability of the programmer to check.

For the previous example, we should have done:

```
ar = malloc(n * sizeof(float));
if (ar == NULL )           /* malloc has failed */
   take-appropriate-action; /* for instance display a
                              message and exit program */
```

9- Passing parameters by address

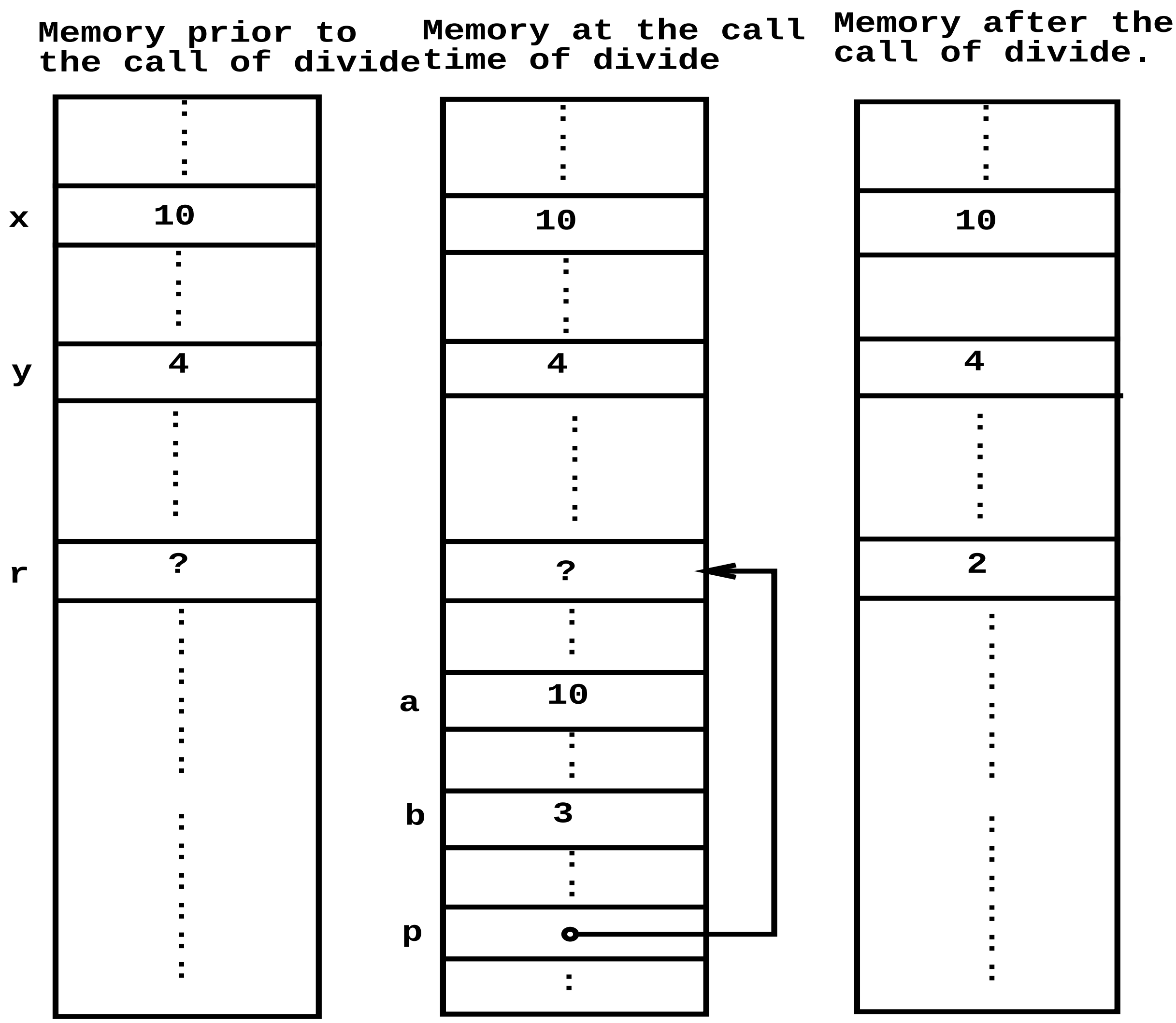
One of the most common application of pointers in C is their use as function parameters, making it possible for a function to get the address of an actual parameters.

By address mode (address parameter): When an argument (parameter) is passed by address to a function, the latter receives the address of the actual argument (the argument put in the "function call" at the call time is a pointer).

Example:

```
int divide(int, int, int *); // divide has 2 VALUE-PARAMETERS of type in
                             // and 1 ADDRESS-PARAMETER of type int
main(){
  int x, y, r;

  printf(``Enter 2 integers please> '');
  scanf(``%d%d'', &x, &y);   // suppose the user enters 10 and 3

  if(divide(x, y, &r))  // if divide returns 1, that is success
    printf(``%d'',r);   // x, y and r are the actual argument. Because
  else                  // x and y are passed by value, their contents w
    printf(``error'');  // copied to the function's arguments a and b
                        // respectively. However, the function's argumen
                        // p will receive the address of the actual
                        // argument r. In particular, r and *p are physi
                        // the same memory cell. That is, *p is
                        // another name for r.
}

int divide(int a, int b, int *p)
{

 if(b==0)
   return(0);           // division by zero
 else
   *p = a/b;            // store the quotient in *p (*p and r are identi

 return(1);             // division successful
}
```

The above example can be better understood when looking inside the memory of the computer during the execution of the program.



Note that a, b and p do not exist prior to the function call and after the function call. p is a pointer variable that has received the address of the actual argument r at the call time.

We can note in particular that :

- what happens to the variables a and b inside the function *divide* WILL NOT affect the variables x and y(the actual argument).
- what happen to the variable $*$p (p is a pointer and $*$p is the variable that contains the data) will affect the variable r since $*$p is simply another name for r.