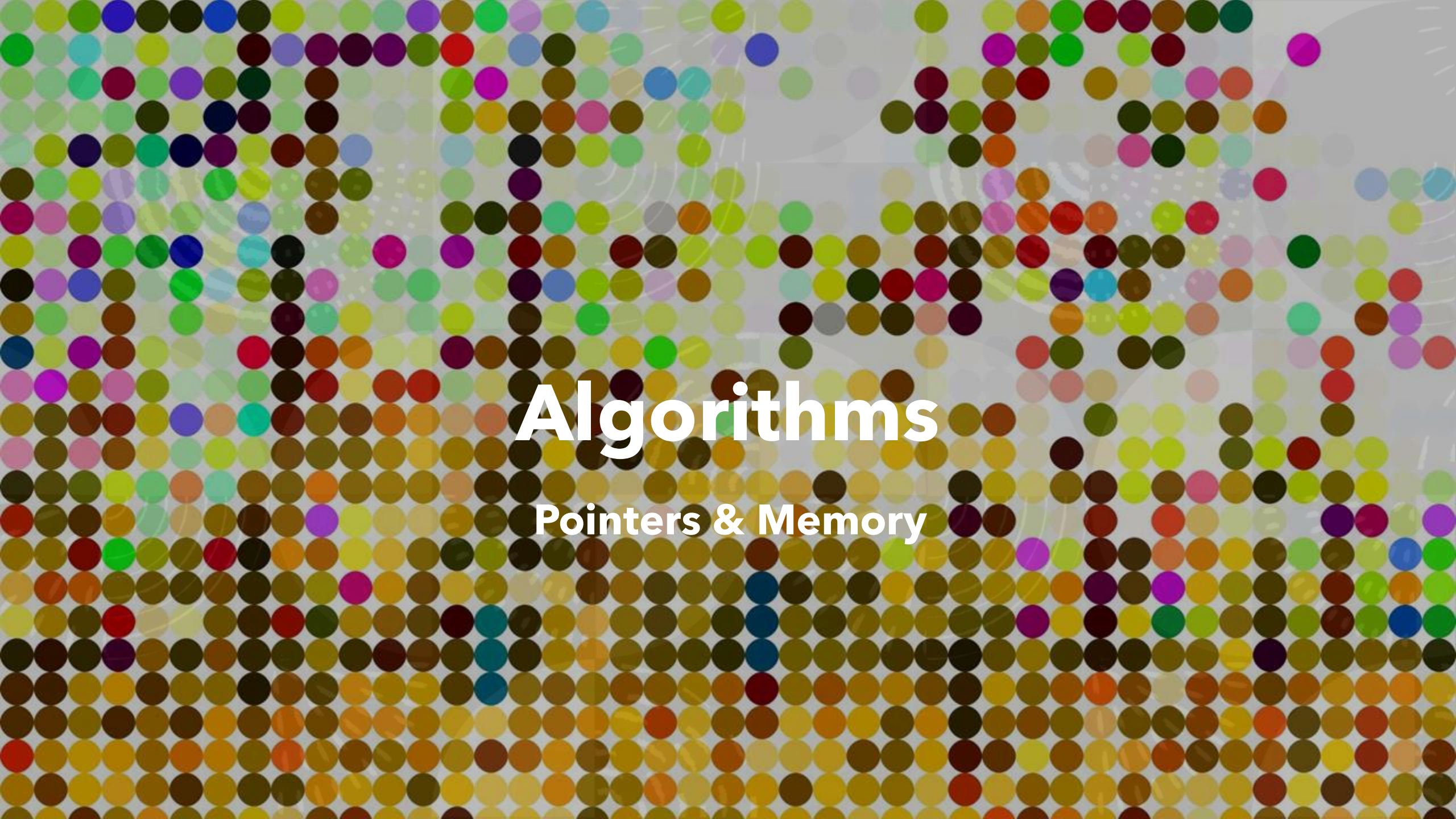

Algorithms
Pointers & Memory

Introduction

Pointers are one of the most powerful aspects of C++ programming, but also one of the most complex to master. **Pointers allow you to manipulate the computer's memory efficiently**. Two concepts are fundamental to understanding how pointers work:

- The size of all variables and their position in memory.

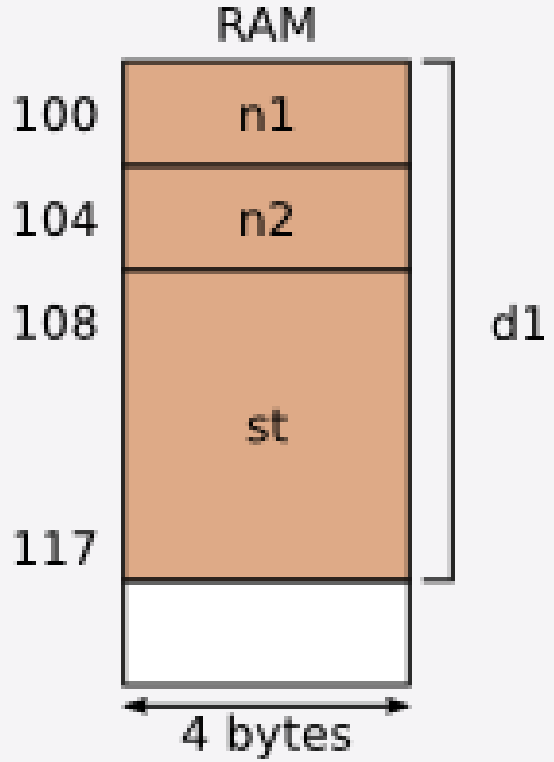
- All data is stored from a memory address. This address can also be obtained and manipulated as another data.

All data has a memory address

All data is stored starting from a memory address and using as many bytes as necessary.

For example, consider the following data definition and its corresponding memory representation starting at address **100**.

```
struct contact
{
    int n1;
    float n2;
    char st[10];
} d1;
```



Assuming that

Char – 1 byte (8 bits)

Int – 4 bytes (32 bits)

Float – 4 bytes (32 bits)

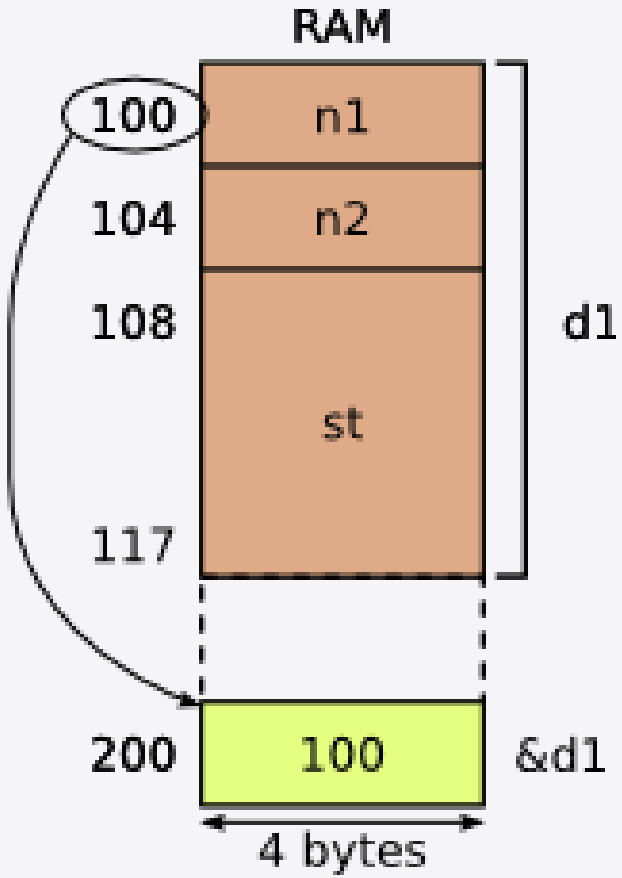
*Data longer than **"word"** length occupies consecutive memory spaces*

Suppose that memory addresses are represented internally with **32 bits** (**4 bytes**).

Then, we can store the address of **d1** (the number **100**) in another place in memory that will occupy **4 bytes** because it is a memory address.

The following figure shows this situation with the memory address stored at location **200**.

```
struct contact
{
    int n1;
    float n2;
    char st[10];
} d1;
```

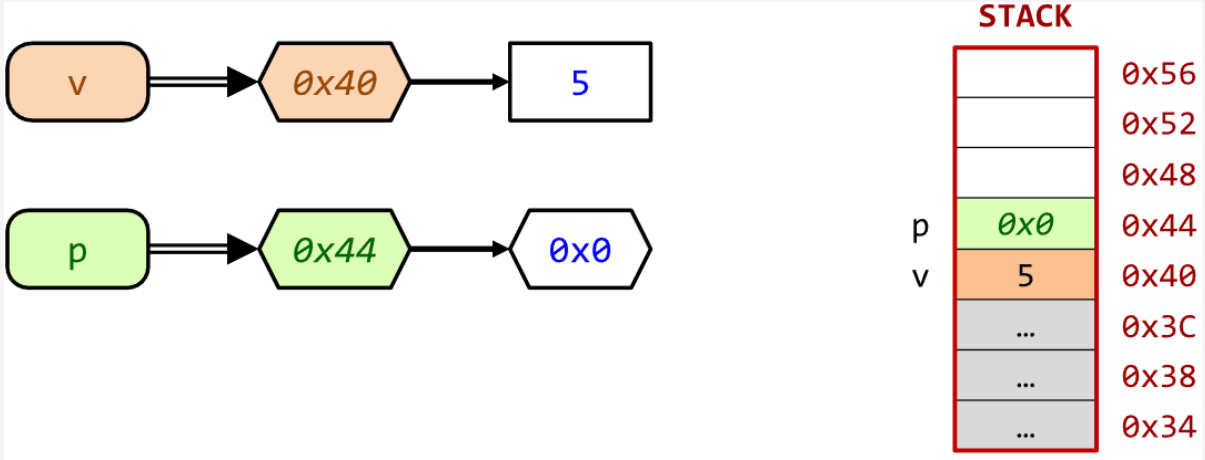


Have you ever wondered why there are 32-bit and 64-bit operating systems?

Pointer declaration

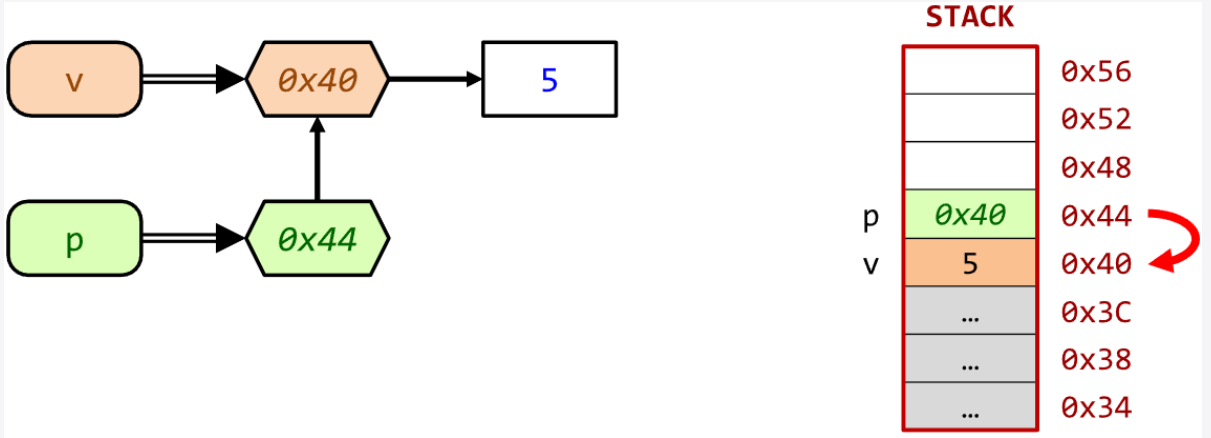
<type> *<indentifier>; e.g., **int *p;**

int *p does not create a variable of **type int** in memory. What creates is a variable that can **contain the memory address of an int**.



To know the memory address of any object, you use the **reference operator** (**&**).

```
int   v = 5;   // value of v is 5
int*  p = &v;  // take address of v
```



If a pointer to the variable **v** is stored at **position 40**, the data of this object can be accessed by an **"indirection"**. The data stored at position 44 (the **number 40**) is now interpreted as an **address**. This address is accessed and from there the fields of **v** are accessed.

The "pointer to" data type

Tipo T	Tamaño (bytes) [a]	Puntero a T	Tamaño (bytes)	Ejemplo de uso
int	4	int *	4	int *a, *b, *c;
unsigned int	4	unsigned int *	4	unsigned int *d, *e, *f;
short int	2	short int *	4	short int *g, *h, *i;
unsigned short int	2	unsigned short int *	4	unsigned short int *j, *k, *l;
long int	4	long int *	4	long int *m, *n, *o;
unsigned long int	4	unsigned long int *	4	unsigned long int *p, *q, *r;
char	1	char *	4	char *s, *t;
unsigned char	1	unsigned char *	4	unsigned char *u, *v;
float	4	float *	4	float *w, *x;
double	8	double *	4	double *y, *z;
long double	8	long double *	4	long double *a1, *a2;

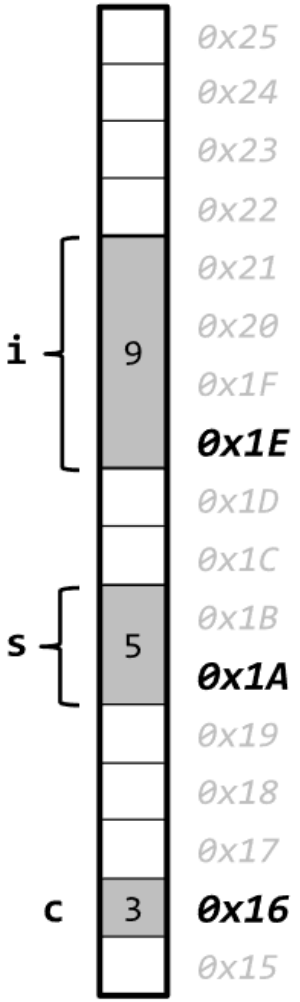
[a] *Reference values, may change depending on the platform.*

The **size of the pointers is always the same** regardless of the data they point to because they all store a memory address.

Pointers Property (1/6)

Data types have different memory sizes

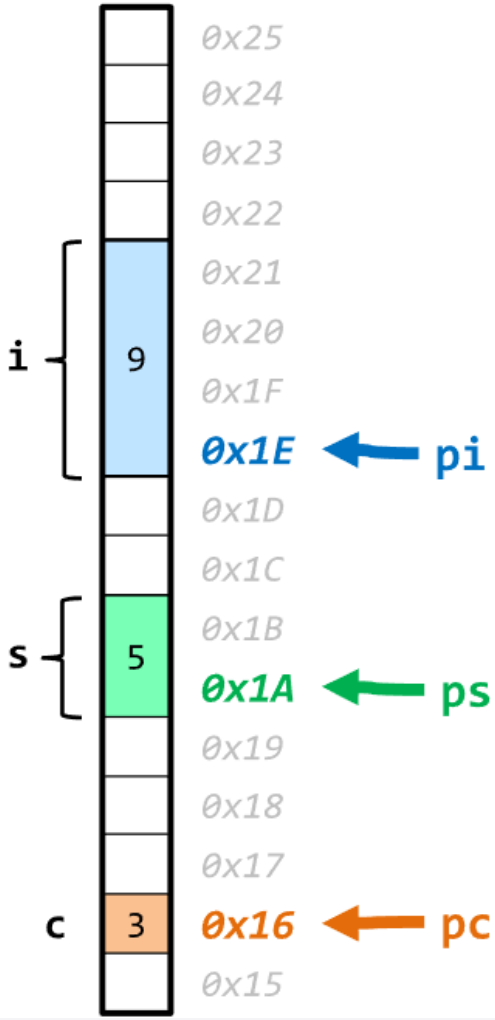
```
char  c = 3;
short s = 5;
int   i = 9;
```



Pointers Property (2/6)

Pointers store **one** address only

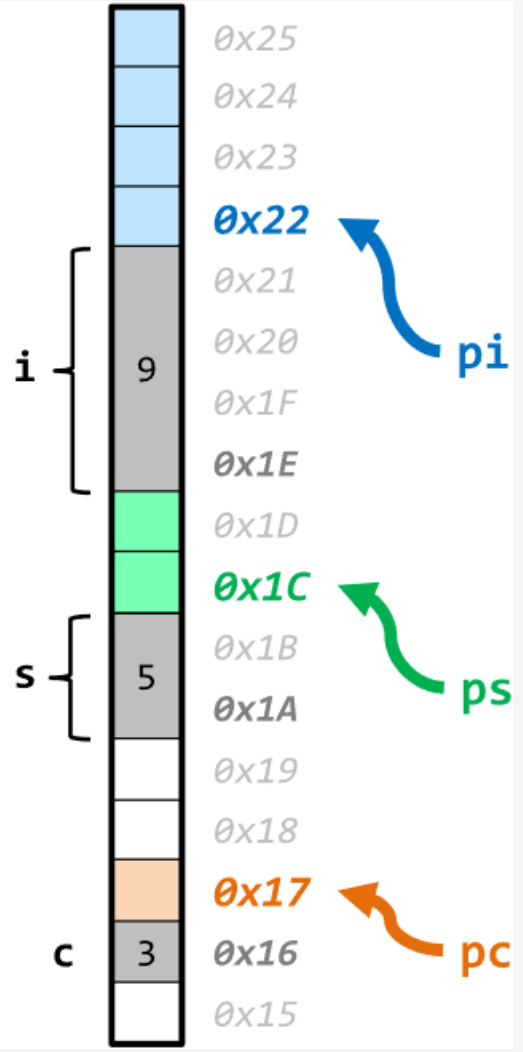
```
char  c = 3;
short s = 5;
int   i = 9;

char*  pc = &c;
short* ps = &s;
int*   pi = &i;
```



Pointers Arithmetic (3/6)

Increment by **1**.

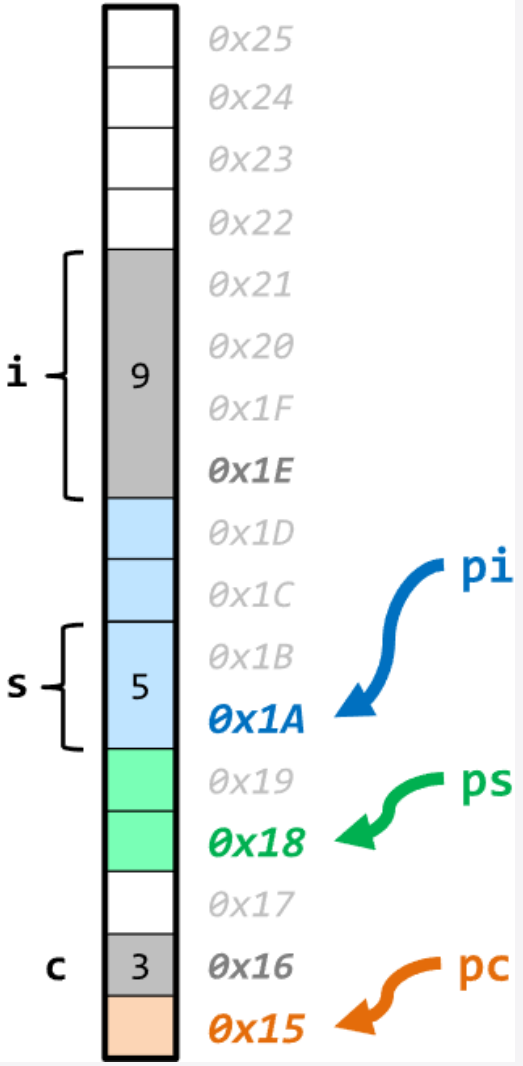
```
char  c = 3;
short s = 5;
int   i = 9;

char*  pc = &c;
short* ps = &s;
int*   pi = &i;

pc++;
ps++;
pi++;
```



Pointers Arithmetic (4/6)

Decrement by **1**.

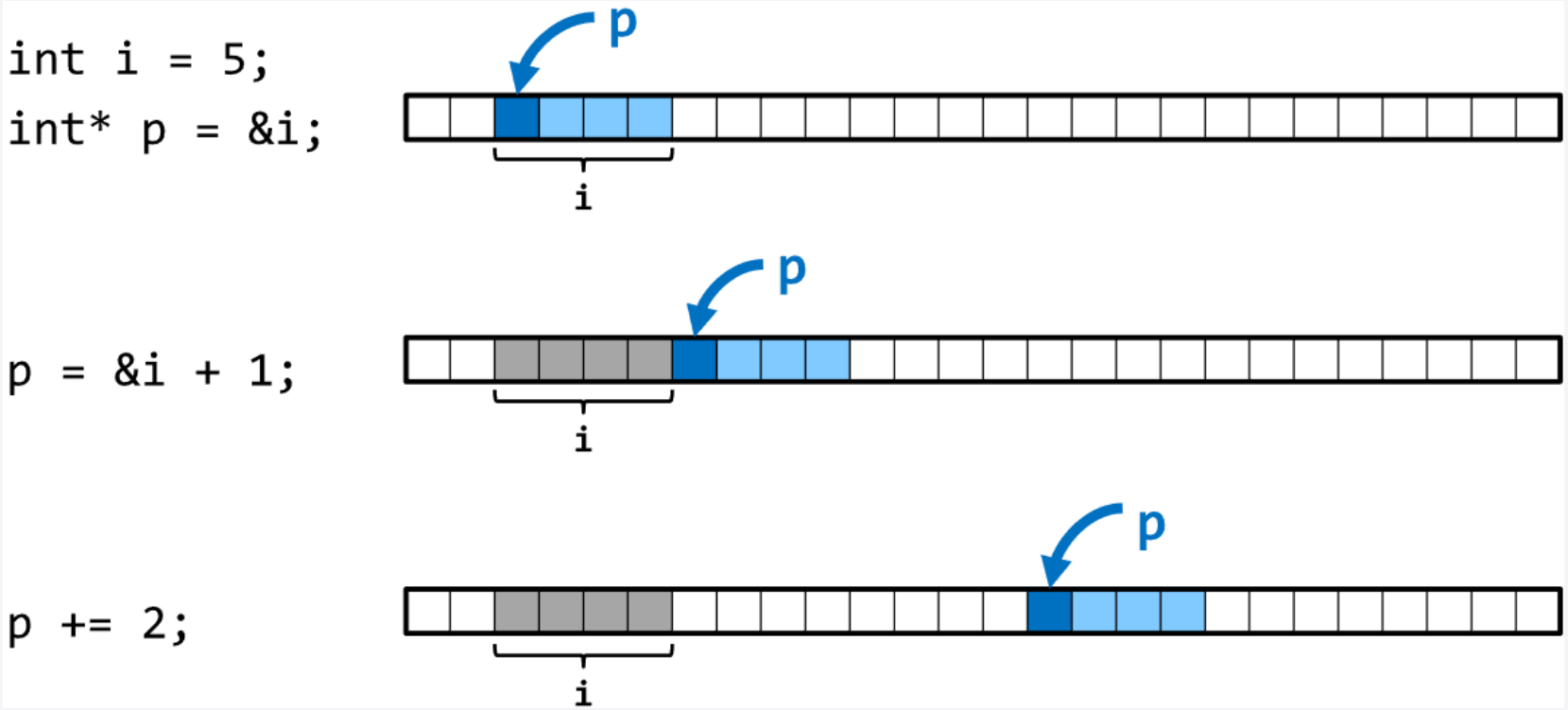
```
char  c = 3;
short s = 5;
int   i = 9;

char*  pc = &c;
short* ps = &s;
int*   pi = &i;

pc--;
ps--;
pi--;
```



Pointers Arithmetic (5/6)

Here:

sizeof(int) = 4 * **sizeof**(char)

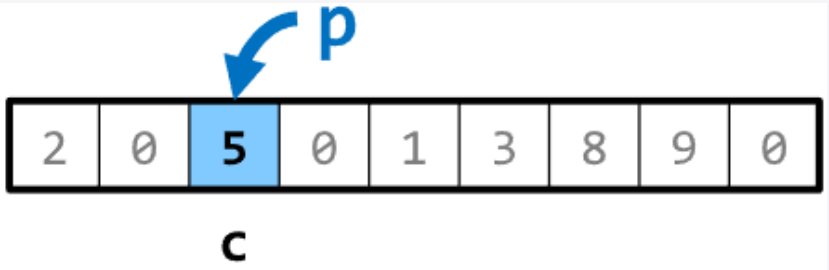


Pointers Arithmetic (6/6)

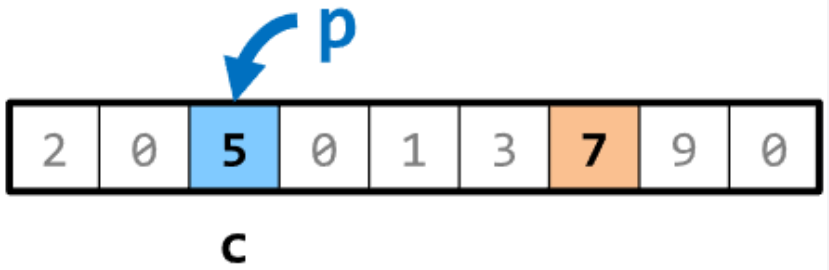
The subscript operator **[]**

P[n] = access value at (address in pointer + **n**)

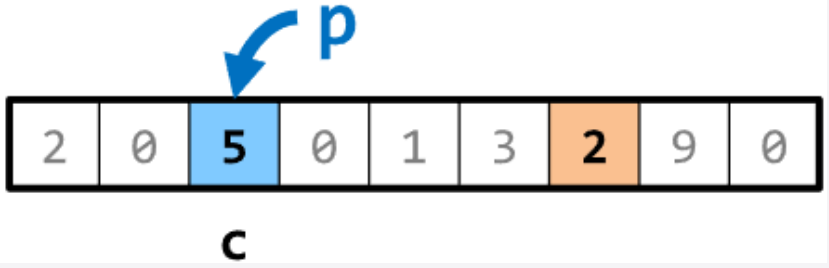
```
char c = 5;
char* p = &c;
```




```
*(p + 4) = 7;
```




```
p[4] = 2;
```



Pointers normal usage

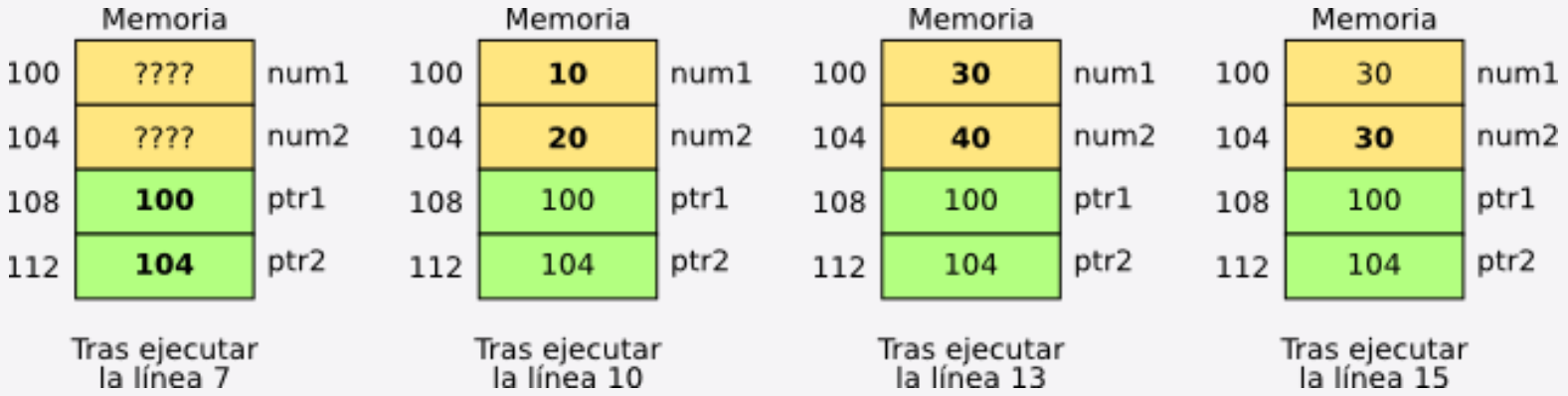
Indirection is denoted by the **operator *** followed by the name of a pointer variable. Its meaning is **"accesses the contents pointed to by the pointer"**.

1. **Line 13** assigns the **value 30** to the memory address stored in **ptr1**. As the pointer has the address of **num1** (line 7) this line is analogous to assigning the **value 30** directly to **num1**.
2. Similarly, **line 14** assigns the **value 40** to the address pointed to by **ptr2**. This assignment is equivalent to assigning the **value 40** to **num2**.
3. **Line 16** contains two indirections. The expression to the right of the equal sign gets **the data at the position** indicated by the address stored in **ptr1** (num1) and this **data is stored at the position** indicated by the pointer **ptr2** (num2).

At the end of the program, the variable **num2** contains the **value 30** even though it has not been directly assigned.

```
1   #include <stdio.h>
2   int main(int argc, char** argv)
3   {
4       int num1, num2;
5       int *ptr1, *ptr2;
6
7       ptr1 = &num1;
8       ptr2 = &num2;
9
10      num1 = 10;
11      num2 = 20;
12
13      *ptr1 = 30;
14      *ptr2 = 40;
15
16      *ptr2 = *ptr1;
17
18      return 0;
19  }
```

```
1  #include <stdio.h>
2  int main(int argc, char** argv)
3  {
4      int num1, num2;
5      int *ptr1, *ptr2;
6
7      ptr1 = &num1;
8      ptr2 = &num2;
9
10     num1 = 10;
11     num2 = 20;
12
13     *ptr1 = 30;
14     *ptr2 = 40;
15
16     *ptr2 = *ptr1;
17
18     return 0;
19 }
```



The **overlap** between the **indirection operator** and the **pointer type definition operator "*"** can make the code difficult to understand. For example, the following lines

```
int m = 20;
int *p = &m;
int n = *p;  //n = 20
```

contains two **"*"** operands. The first one defines the type **"pointer to integer"** and is followed by the declaration of the **variable "p"**. The second is an **indirection** that is applied to the address stored in the **pointer "p"**.

The concepts of **value**, **pointer**, and **reference** often trip people up.

- A **value** is a thing that exists and has meaning. It always has a type like *int*, *float*, *char*, or *vector<string>*.

- A **reference** is an alias – a name for an object. The name is convenient for us as programmers. If you have a name, you have direct access to an object.

- A **pointer** is a value. Its type is a location in the computer's memory of some object. With this location, I can gain access to the object's value. In other words, I can use a pointer to gain a direct reference to an object.

Pointers to an Array

As a principle the name of an array is a constant pointer to the first element of the array. That is, in

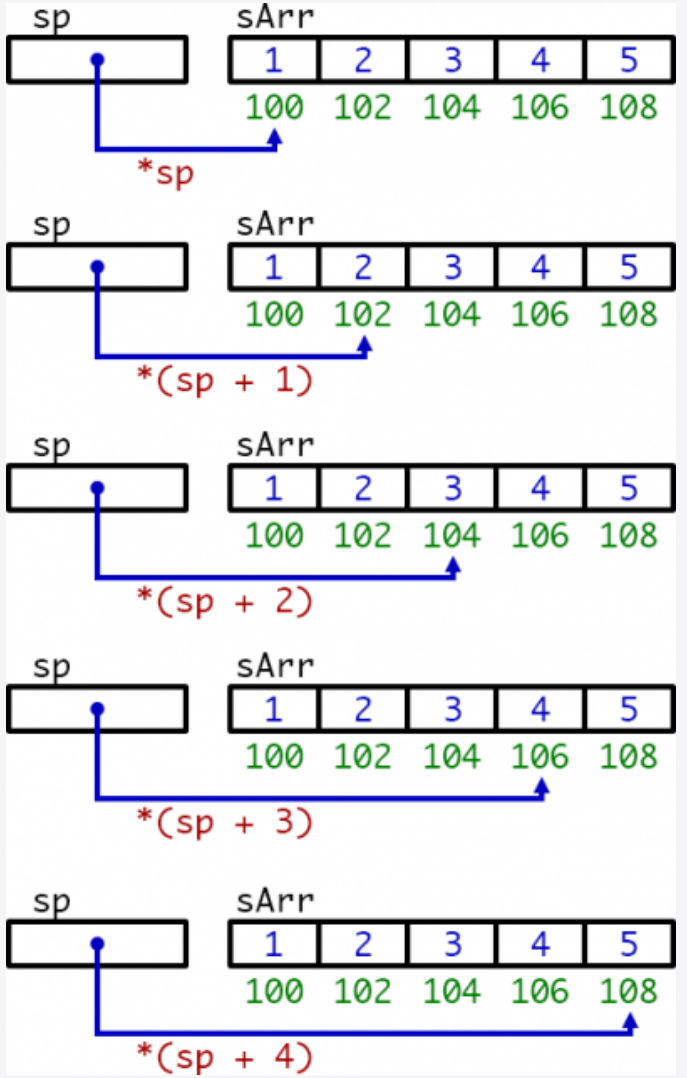
int sArr[5] = {1, 2, 3, 4, 5}

sArr is a pointer to **&sArr[0]** which is the address of the first element of the array. Therefore using a pointer to manipulate an array makes sense.

Int *sp;

sp = sArr;

Since arrays occupy consecutive memory spaces, we can access all the data in the array.

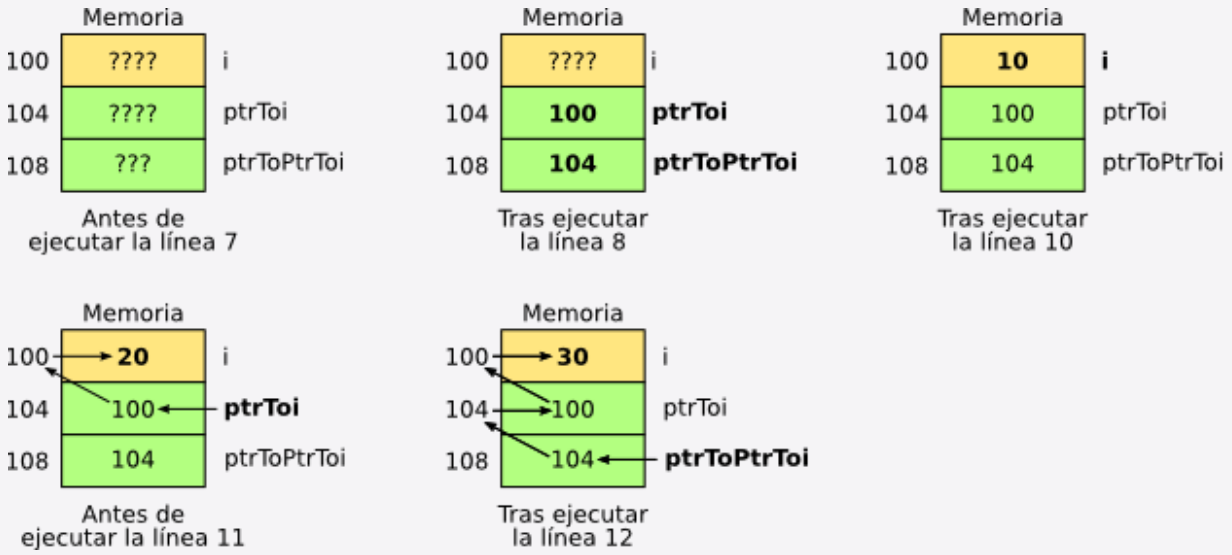


Pointers to Pointers

The derivation rule whereby from a **type T** is derived the type **"pointer to T"** can also be applied to this new pointer to obtain the type **"pointer to pointer to T"** defined as **"T **"**.

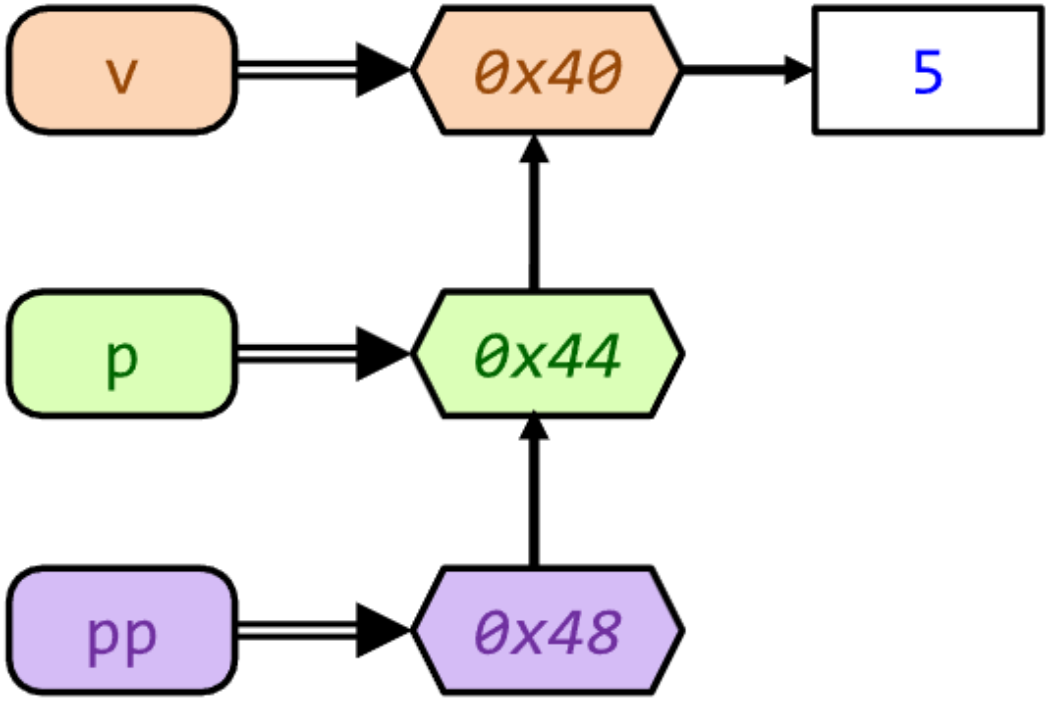
These multiple pointers have the same size as a single pointer, the difference is only in the number of indirections to obtain the data.

```c
#include <stdio.h>
int main()
{
    int i;
    int *ptrToi;            /* Puntero a entero */
    int **ptrToPtrToi;      /* Puntero a puntero a entero */

    ptrToPtrToi = &ptrToi; /* Puntero contiene dirección de puntero */
    ptrToi = &i;            /* Puntero contiene dirección de entero */

    i = 10;                 /* Asignación directa */
    *ptrToi = 20;           /* Asignación indirecta */
    **ptrToPtrToi = 30;     /* Asignación con doble indirección */

    return 0;
}
```



Pointers to Pointers – Example 2

```
int    v = 5;
int*   p = &v;
int** pp = &p;

cout << v;      // 5
cout << p;      // 0x40
cout << pp;     // 0x44

cout << &v;     // 0x40 (= p)
cout << &p;     // 0x44 (= pp)
cout << &pp;    // 0x48

cout << *p;     // 5
cout << *pp;    // 0x40 (= p)
cout << **pp;   // 5
```



Parameters by reference

When a variable is **passed by reference**, the compiler **does not pass a copy of a value of the argument**; it passes a **reference** that tells the function **where the variable is in memory**. Therefore, by passing the memory address, **when changes are made to that variable in the function, the value of the variable will be changed and not the value of its copy**.

```c
#include <stdio.h>

     /* Definición de función "swap". Fíjate que las variables se reciben como puntero a esas variables. */
    void sswap ( int *x, int *y )
     {
         /*Declaramos una variable temporal*/
         int tmp;
         tmp = *x;
         *x = *y;
         *y = tmp;
     }

     int main()
     {
           int a, b;
           a = 1;
           b = 2;

           /*Llamamos a la función "swap" pasándole la dirección a las variables a y b.*/
           swap( &a, &b );
           /*Imprime los valores de a y b intercambiados*/
           printf(" a = %d b = %d\n", a, b );

     }
```

In the **pass-by-value** case, where a copy of the data is passed. This copy has two effects: the program spends as much time as there is data in the structure, and during the execution of the function two complete copies of the structures are kept in memory. The use of pointers offers a more efficient alternative.

```
struct coordenadas
{
    float x;
    float y;
    float z;
};
```

```
float distancia(struct coordenadas *a_ptr, struct coordenadas *b_ptr)
{
    return sqrtf(pow(a_ptr->x - b_ptr->x, 2.0) +
                 pow(a_ptr->y - b_ptr->y, 2.0) +
                 pow(a_ptr->z - b_ptr->z, 2.0));
}
```

Where you call the function in the following way:

```
struct coordenadas punto_a = { 3.5e-120, 2.5, 1.5 };
struct coordenadas punto_b = { 5.3e-120, 3.1, 6.3 };
```

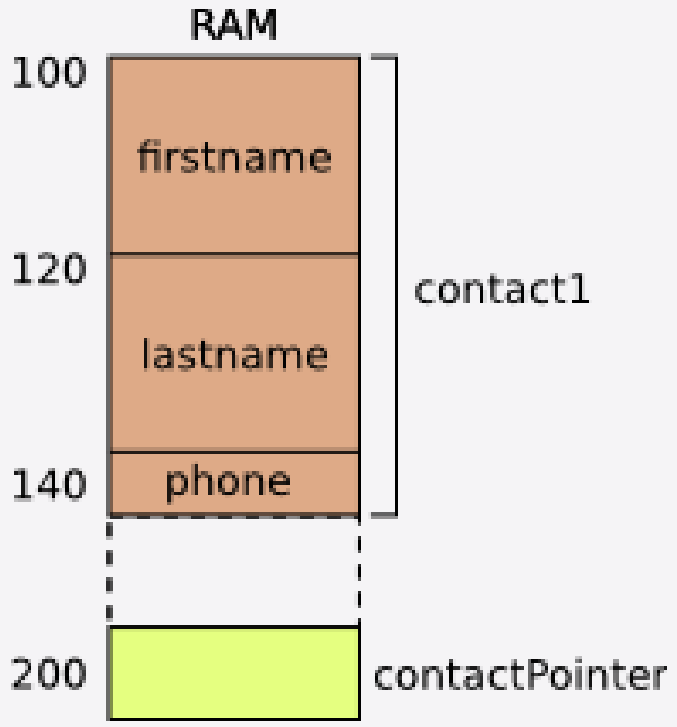
```
d = distancia(&punto_a, &punto_b);
```

With this new version, the program executes exactly the same calculations, obtains the same result, but runs faster and uses less memory.

Pointers to data structures

For **data structures**, the rule applies identically. The following example show the declaration of a variable of type structure and another of type pointer to that structure:

```
struct contact
{
    char firstname[20];
    char lastname[20];
    unsigned int phone;
};

struct contact contact1;

struct contact *contactPointer;
```



The **contact1** variable is of **type structure** and occupies **44 bytes** (20+20+4), while **contact Pointer** is of **type pointer** and occupies **4 bytes**.

Selection operators (. , ->)

When a pointer indirection is performed, it is only to access the variables of the structure. This operation requires two operators, the indirection (**operator "*"**) and the access to the variable of a structure (**operator "."**). The syntax is:

```
(*pointer).x = 10;
(*pointer).c = 'l';
```

These two operations are grouped in the **operator "->"** which is placed between the pointer and the variable name of the structure. The equivalent notation, therefore, is:

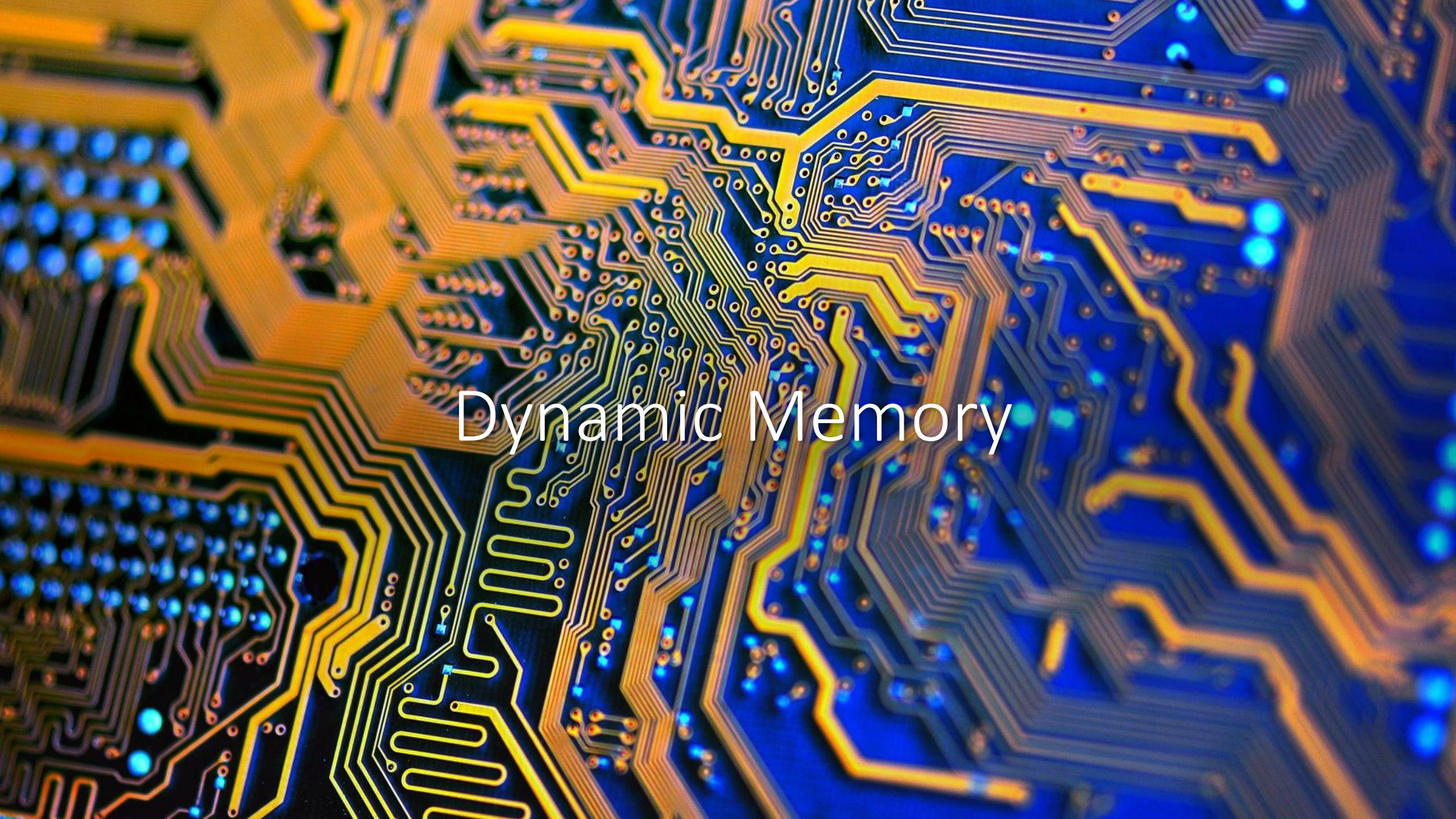
```
pointer -> x = 10;
pointer -> c = 'l';
```

```
1  struct s
2  {
3      int x;
4      char c;
5  }
6
7  struct s element;
8  struct s *pointer;
9
10 pointer = &element;
```

The **operator "->"** is always written after a pointer pointing to a structure, and precedes the name of one of the variables of that structure.

The following program shows how data can be copied from one structure to another using this operator.

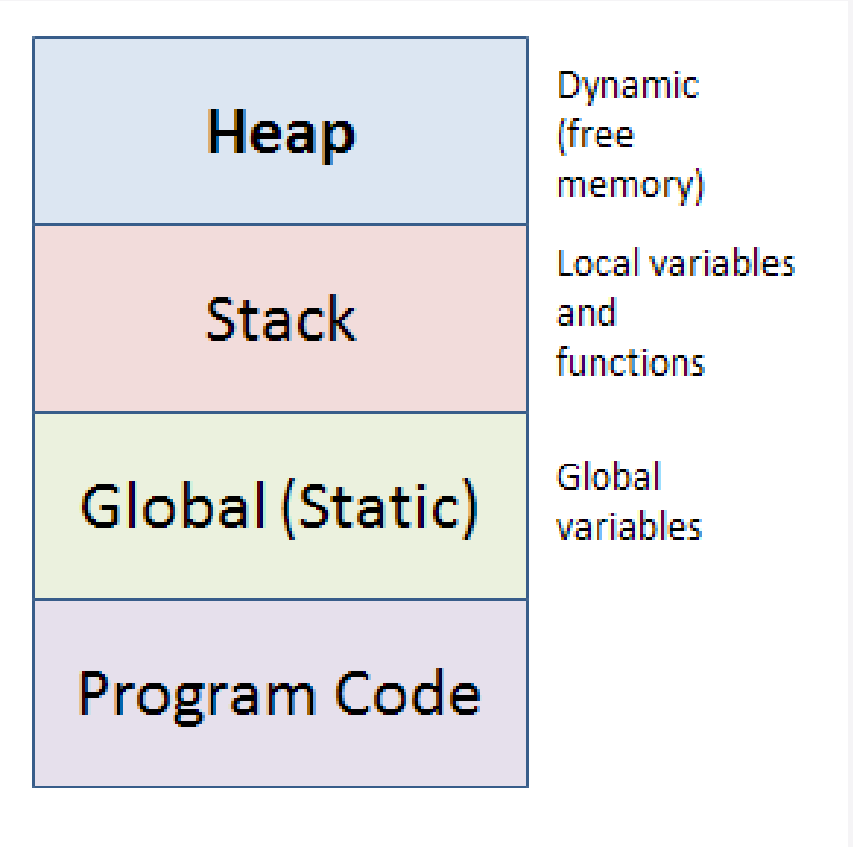
```c
/* Definición de la estructura */
struct coordenadas
{
    float x;
    float y;
    float z;
};

int main()
{
    /* Declaración de dos estructuras */
    struct coordenadas location1, location2;
    /* Declaración de dos punteros */
    struct coordenadas *ptr1, *ptr2;

    /* Asignación de direcciones a los punteros */
    ptr1 = &location1;
    ptr2 = &location2;

    /* Asignación de valores a la primera estructura */
    ptr1->x = 3.5;
    ptr1->y = 5.5;
    ptr1->z = 10.5;

    /* Copia de valores a la segunda estructura */
    ptr2->x = ptr1->x;
    ptr2->y = ptr1->y;
    ptr2->z = ptr1->z;

    return 0;
}
```


Dynamic Memory

Memory management

Memory is classified into many categories. The operating system reserves one area of memory for program code or instructions and another for local variables that are used during runtime (**stack**).

The rest of the memory that is not used by any program is known as **Heap**. Our program can make use of the heap to vary the size of our application at runtime, that is why it is called **dynamic memory**.

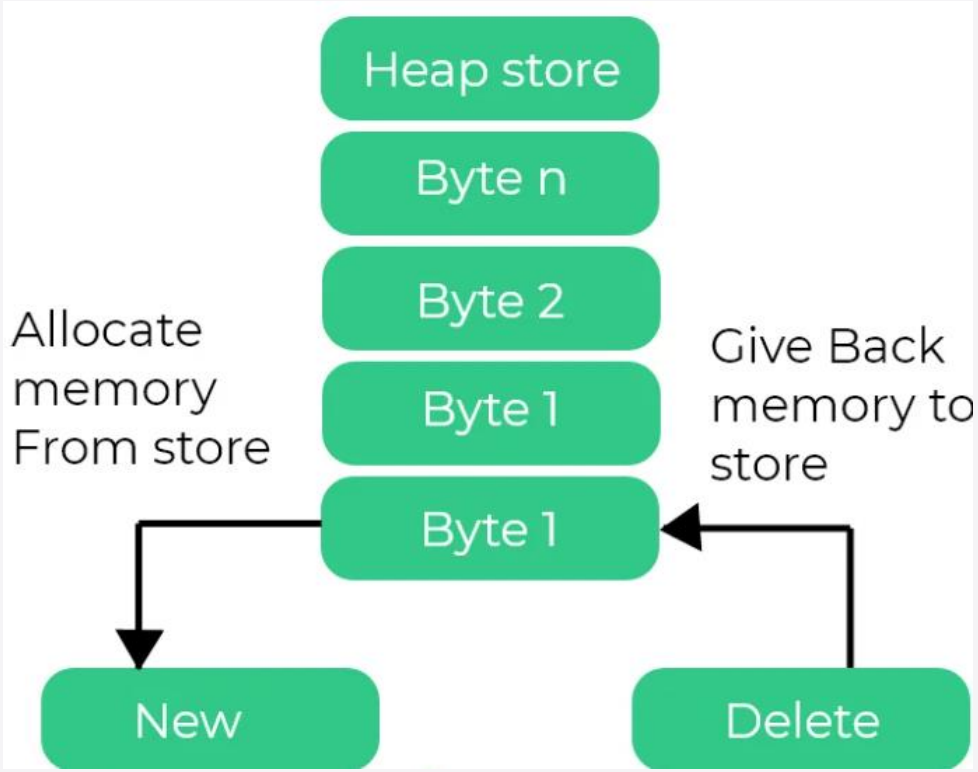


Operator "new" & "delete"

C++ has two operators to reserve and free dynamic memory, they are **"new"** and **"delete"**. All reserved memory must be freed before exiting the program.

The **malloc()** function from C, still exists in C++, but **it is recommended to avoid using malloc()** function.

The main advantage of **new** over **malloc()** is that **new** doesn't just **allocate memory**, it **constructs objects** which is prime purpose of C++.



"new" Operator

The generic syntax to use **new** operator to allocate memory dynamically for any data-type.

pointer = **new** <data_type>;

or

pointer = **new** <data_type> [number_of_elements];

Here, data-type could be any built-in data type including an **array** or any user defined data types include **class** or **structure**.

For example, we can define a **pointer** to type **double** and then request that the memory be allocated at execution time.

```
double* pvalue  = NULL; // Pointer initialized with null
pvalue  = new double;   // Request memory for the variable
```

"delete" Operators

At any point, when you feel a variable that has been dynamically allocated is not anymore required, you can free up the memory that it occupies in the free store with the **"delete"** operator as follows –

```
double* pvalue  = NULL; // Pointer initialized with null
pvalue  = new double;   // Request memory for the variable
```

```
delete pvalue;        // Release memory pointed to by pvalue
```

Example 2

```
#include <iostream>
using namespace std;

int main () {
   double* pvalue  = NULL; // Pointer initialized with null
   pvalue  = new double;   // Request memory for the variable

   *pvalue = 29494.99;     // Store value at allocated address
   cout << "Value of pvalue : " << *pvalue << endl;

   delete pvalue;         // free up the memory.

   return 0;
}
```

If we compile and run above code, this will produce the following result

```
Value of pvalue : 29495
```

Dynamic Memory Allocation for Arrays

Consider you want to allocate memory for an **array** of characters, i.e., string of 20 characters.

```
char* pvalue  = NULL;          // Pointer initialized with null
pvalue  = new char[20];        // Request memory for the variable
```

To remove the array that we have just created the statement would look like this

```
delete [] pvalue;              // Delete array pointed to by pvalue
```

Following the similar generic syntax of **new** operator, you can allocate for a **fixed size multi-dimensional array** as follows

```
double** pvalue  = NULL;      // Pointer initialized with null
pvalue  = new double [3][4];  // Allocate memory for a 3x4 array
```

However, the syntax to release the memory for multi-dimensional array will still remain same as above

```
delete [] pvalue;             // Delete array pointed to by pvalue
```

However, if you had a **dynamically allocate multi-dimensional array** as follows

```
//Create an array of pointers that points to more arrays
int** matrix = new int*[5];
for (int i = 0; i < 5; ++i) {
    matrix[i] = new int[5];
    for (int j = 0; j < 5; ++j) {
        matrix[i][j] = i*5 + j;
    }
}
```

The recommendation to release the memory will be

```
//Free each sub-array
for(int i = 0; i < 5; ++i) {
    delete[] matrix[i];
}
//Free the array of pointers
delete[] matrix;
```

Dynamic Memory Allocation for Objects

Objects are no different from simple data types. Consider the following code where we are going to use an **array of objects**:

```
#include <iostream>
using namespace std;

class Box {
   public:
      Box() {
         cout << "Constructor called!" <<endl;
      }
      ~Box() {
         cout << "Destructor called!" <<endl;
      }
};
int main() {
   Box* myBoxArray = new Box[4];
   delete [] myBoxArray; // Delete array

   return 0;
}
```

If you were to allocate an array of **four Box objects**, the **Simple constructor would be called four times** and similarly while deleting these objects, **destructor will also be called same number of times**.

If we compile and run above code, this will produce the following result:

```
Constructor called!
Constructor called!
Constructor called!
Constructor called!
Destructor called!
Destructor called!
Destructor called!
Destructor called!
```

```
#include <iostream>
using namespace std;

class Box {
   public:
      Box() {
         cout << "Constructor called!" <<endl;
      }
      ~Box() {
         cout << "Destructor called!" <<endl;
      }
};
int main() {
   Box* myBoxArray = new Box[4];
   delete [] myBoxArray; // Delete array

   return 0;
}
```