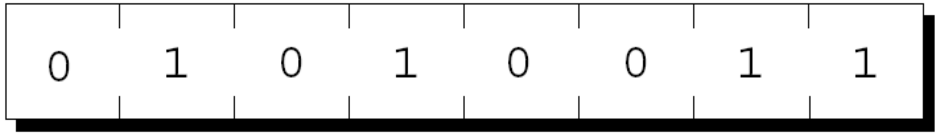
Elementi Di Informatica E Programmazione

Prof. Andrea Loreggia

UNIVERSITÀ DEGLI STUDI DI BRESCIA

Variabili puntatore

- Il primo passo per comprendere i puntatori è visualizzare cosa rappresentano a livello macchina.
- Nella maggior parte dei computer moderni, la memoria principale è divisa in byte, con ciascun byte in grado di memorizzare otto bit di informazioni:



- Ogni byte ha un indirizzo univoco.

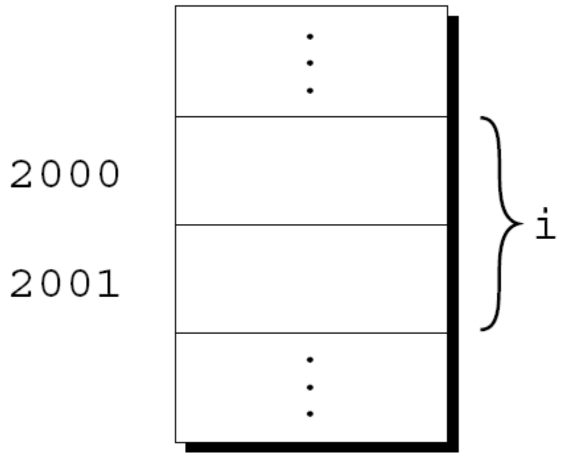
Variabili puntatore

- Se ci sono n byte in memoria, possiamo pensare agli indirizzi come numeri che vanno da 0 a n - 1.

Address	Contents
0	01010011
1	01110101
2	01110011
3	01100001
4	01101110
	⋮
n-1	01000011

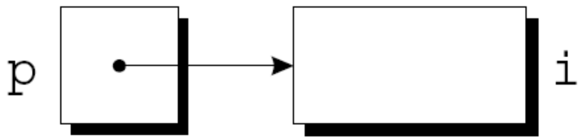
Variabili puntatore

- Ogni variabile in un programma occupa uno o più byte di memoria.
- L'indirizzo del primo byte è considerato l'indirizzo della variabile.
- Nella figura seguente, l'indirizzo della variabile i è 2000:



Variabili puntatore

- Gli indirizzi possono essere memorizzati in speciali variabili puntatore. Quando memorizziamo l'indirizzo di una variabile i nella variabile puntatore p, diciamo che p "punta a" i. Una rappresentazione grafica:



Dichiarazione di variabli puntatore

- Quando viene dichiarata una variabile puntatore, il suo nome deve essere preceduto da un asterisco:

```
int *p;
```

- La variabile p è un puntatore capace di puntare a oggetti di tipo int. Usiamo il termine "oggetto" invece di "variabile" perché p potrebbe puntare a un'area di memoria che non appartiene a una variabile specifica.

Dichiarazione di variabli puntatore

UNIVERSITÀ DEGLI STUDI DI BRESCIA

- Le variabili puntatore possono comparire nelle dichiarazioni insieme ad altre variabili. Ad esempio:

```
int i, j, a[10], b[20], *p, *q;
```

- In C, ogni variabile puntatore deve puntare solo a oggetti di un tipo specifico (il tipo riferito):

```
int *p;      /* points only to integers   */
double *q;   /* points only to doubles    */
char *r;     /* points only to characters */
```

- Non ci sono restrizioni su quale possa essere il tipo riferito da un puntatore, purché sia un tipo valido in C.

Operatori di indirizzo e indirezione

- In C, ci sono due operatori specificamente progettati per l'uso con i puntatori:
  1. **Operatore & (indirizzo)**: Per trovare l'indirizzo di una variabile, utilizziamo l'operatore & (indirizzo).
  2. **Operatore * (indirezione)**: Per accedere all'oggetto a cui punta un puntatore, utilizziamo l'operatore * (indirezione).

The Address Operator

- La dichiarazione di una variabile puntatore riserva spazio per un puntatore ma non lo fa puntare a un oggetto specifico:

```
int *p;  /* points nowhere in particular */
```

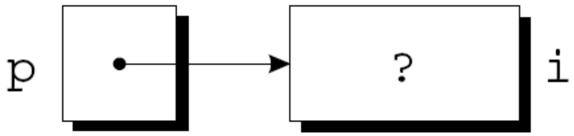
- È fondamentale inizializzare p prima di utilizzarlo per accedere o manipolare dati. Se un puntatore non viene inizializzato, il suo valore iniziale sarà indeterminato e potrebbe causare comportamenti imprevedibili quando si tenta di utilizzarlo per accedere a dati in memoria.

The Address Operator

- Per inizializzare un puntatore, è possibile assegnargli l'indirizzo di una variabile esistente o utilizzare NULL se non si desidera farlo puntare a nulla:

```c
int i, *p;
…
p = &i;
```

- Assegnare l'indirizzo di una variabile i alla variabile puntatore p fa sì che p punti a i.



The Indirection Operator

- Una volta che una variabile puntatore punta a un oggetto, possiamo utilizzare l'operatore * (indirezione) per accedere a ciò che è memorizzato nell'oggetto puntato.
- Se p punta a i, possiamo stampare il valore di i come segue:

```
printf("%d\n", *p);
```

- Applicare & a una variabile produce un puntatore alla variabile stessa. Applicando * al puntatore, torniamo all'oggetto originale:

```
j = *&i;    /* same as j = i; */
```

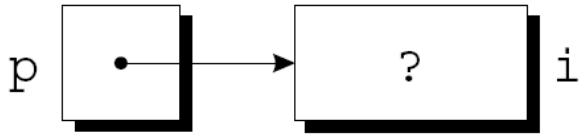
The Indirection Operator

- Finché p punta a i, *p è un alias per i.
  - `*p` ha lo stesso valore di `i`.
  - Cambiare il valore di *p cambia il valore di i.

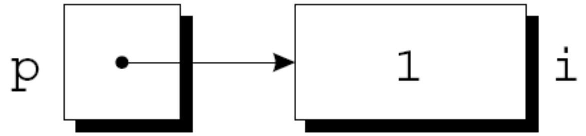
- L'esempio nella prossima diapositiva illustra l'equivalenza di *p e i.

The Indirection Operator

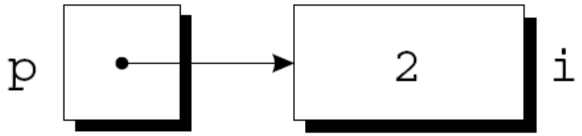
```c
p = &i;
```




```c
i = 1;
```




```c
printf("%d\n", i);     /* prints 1 */
printf("%d\n", *p);    /* prints 1 */
*p = 2;
```




```c
printf("%d\n", i);     /* prints 2 */
printf("%d\n", *p);    /* prints 2 */
```

The Indirection Operator

UNIVERSITÀ DEGLI STUDI DI BRESCIA

- Applicare l'operatore di indirezione a una variabile puntatore non inizializzata causa un comportamento non definito:

```
int *p;
printf("%d", *p);    /*** WRONG ***/
```

- Assegnare un valore a *p è particolarmente pericoloso nel caso in cui il puntatore p non sia stato inizializzato:

```
int *p;    /*p non yet initialized */
*p = 1;    /*** WRONG ***/
```

Pointer Assignment

UNIVERSITÀ DEGLI STUDI DI BRESCIA

- In C, è consentito l'uso dell'operatore di assegnazione per copiare puntatori dello stesso tipo. Assumiamo che la seguente dichiarazione sia in vigore:

```
int i, j, *p, *q;
```

- Esempio di assegnazione di puntatore:

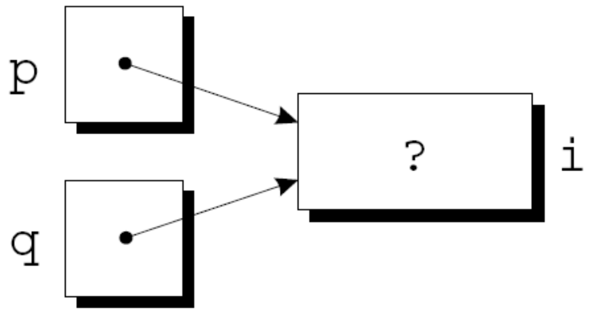
```
p = &i;
```

Pointer Assignment

- Un altro esempio di assegnazione di puntatore:

```
q = p;
```

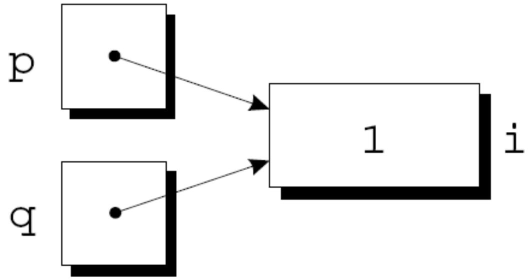
- `q` e `p` adesso puntano entrambi a i:



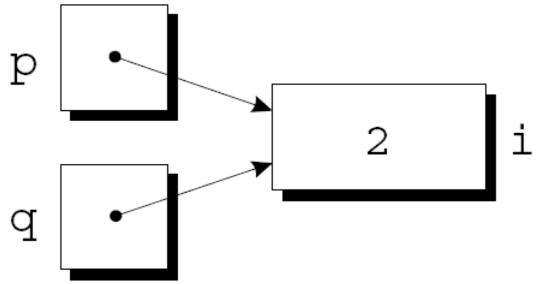
Pointer Assignment

- Se p e q puntano entrambi a i, possiamo modificare i assegnando un nuovo valore sia a *p che a *q:

```
*p = 1;
```



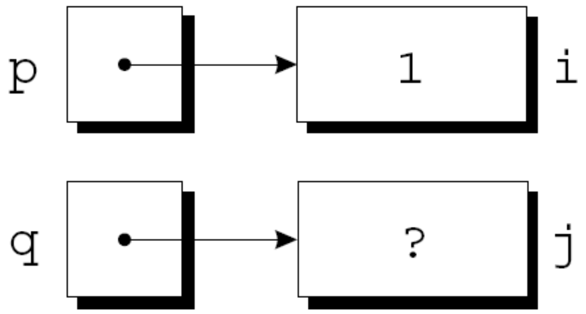

```
*q = 2;
```



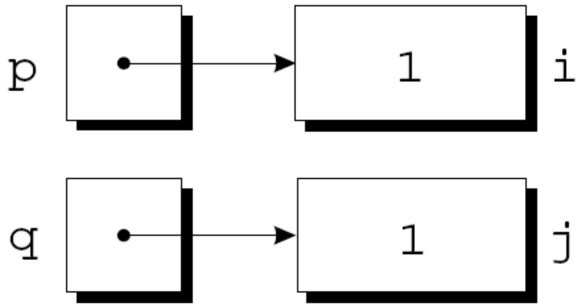
- In C, è possibile avere qualsiasi numero di variabili puntatore che puntano allo stesso oggetto

ATTENZIONE: Pointer Assignment

```
p = &i;
q = &j;
i = 1;
```




```
*q = *p;
```



Program: Finding the Largest and Smallest Elements in an Array

- `max_min.c` will read 10 numbers into an array, pass it to the `max_min` function, and print the results:

```
Enter 10 numbers: 34 82 49 102 7 94 23 11 50 31
Largest: 102
Smallest: 7
```

maxmin.c

```
/* Finds the largest and smallest elements in an array
*/
#include <stdio.h>
#define N 10

void max_min(int a[], int n, int *max, int *min);

int main(void)
{
        int b[N], i, big, small;

        printf("Enter %d numbers: ", N);
        for (i = 0; i < N; i++)
                scanf("%d", &b[i]);

        max_min(b, N, &big, &small);

        printf("Largest: %d\n", big);
        printf("Smallest: %d\n", small);

        return 0;
}
```

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

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

Introduction

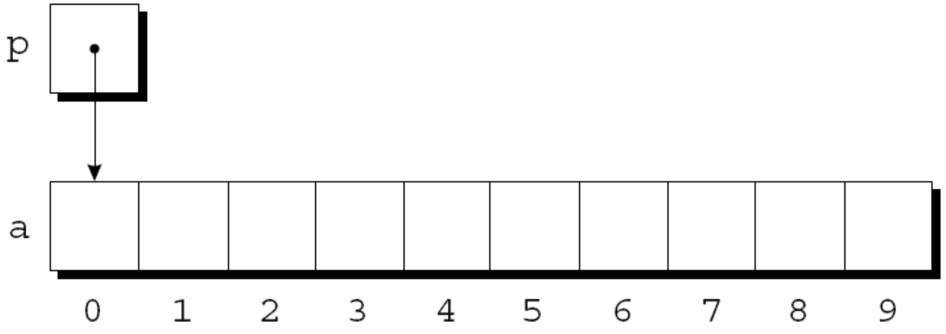
- C allows us to perform arithmetic—addition and subtraction—on pointers to array elements.
- This leads to an alternative way of processing arrays in which pointers take the place of array subscripts.
- The relationship between pointers and arrays in C is a close one.
- Understanding this relationship is critical for mastering C.

Pointer Arithmetic

- Pointers can point to array elements:

```
int a[10], *p;
p = &a[0];
```

- A graphical representation:

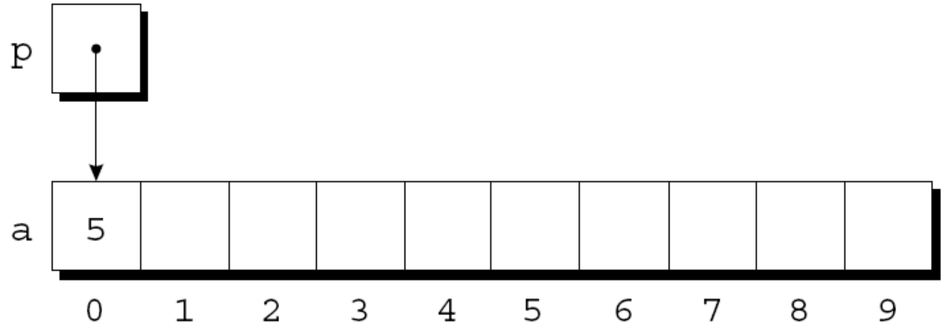


Pointer Arithmetic

- We can now access `a[0]` through `p`; for example, we can store the value 5 in `a[0]` by writing

```
*p = 5;
```

- An updated picture:



Pointer Arithmetic

- If `p` points to an element of an array `a`, the other elements of `a` can be accessed by performing ***pointer arithmetic*** (or ***address arithmetic***) on `p`.
- C supports three (and only three) forms of pointer arithmetic:
  - Adding an integer to a pointer
  - Subtracting an integer from a pointer
  - Subtracting one pointer from another

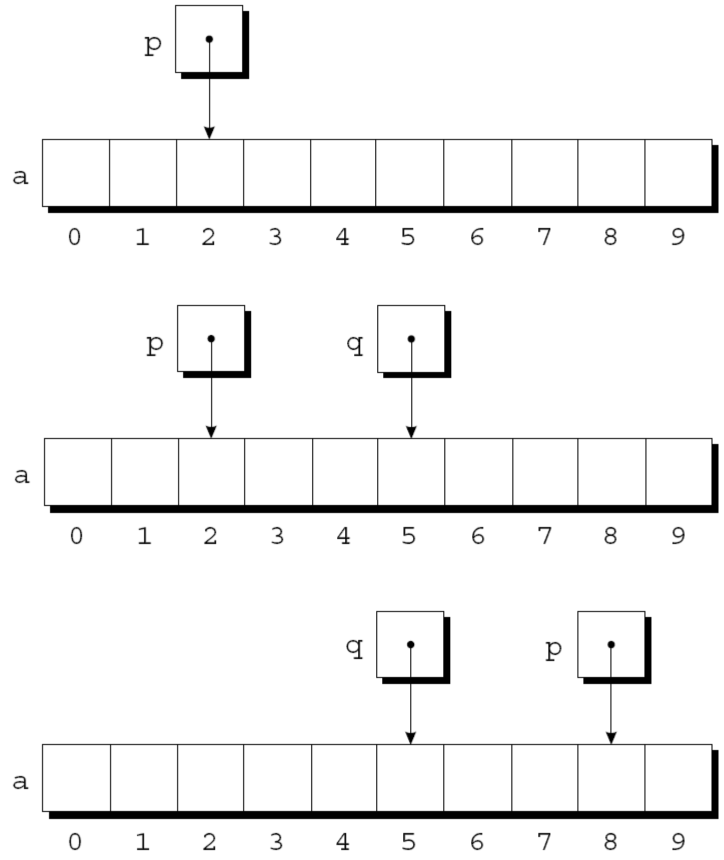
Adding an Integer to a Pointer

- Example of pointer addition:

```
p = &a[2];
```

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

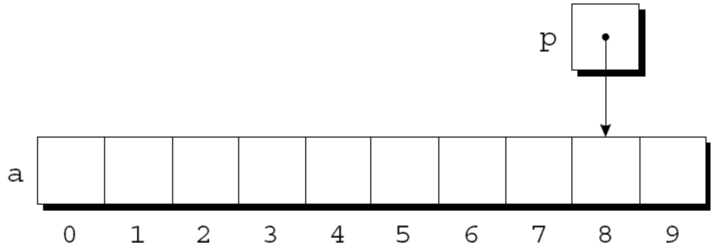
```
p += 6;
```



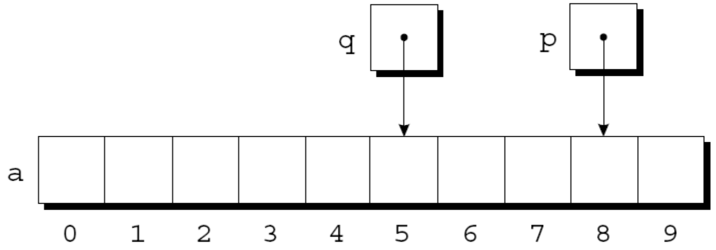
Subtracting an Integer from a Pointer

- If `p` points to `a[i]`, then `p - j` points to `a[i-j]`.
- Example:

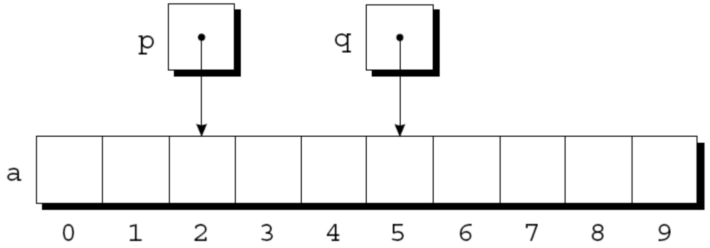
```
p = &a[8];
```




```
q = p - 3;
```



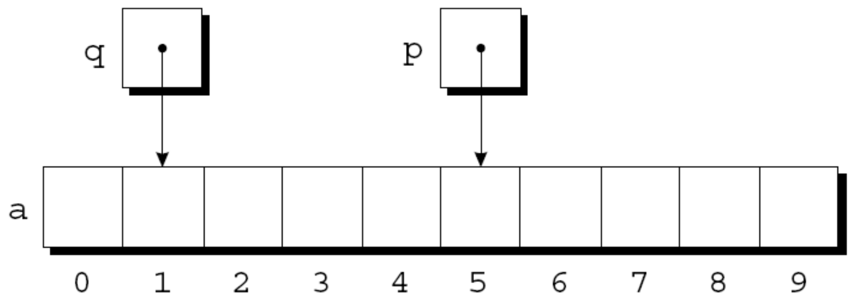

```
p -= 6;
```



Subtracting One Pointer from Another

- When one pointer is subtracted from another, the result is the distance (measured in array elements) between the pointers.
- If `p` points to `a[i]` and `q` points to `a[j]`, then `p - q` is equal to `i - j`.
- Example:

```
p = &a[5];
q = &a[1];
```




```
i = p - q;    /* i is 4 */
i = q - p;    /* i is -4 */
```

Subtracting One Pointer from Another

- Operations that cause undefined behavior:
  - Performing arithmetic on a pointer that doesn’t point to an array element
  - Subtracting pointers unless both point to elements of the same array

Comparing Pointers

- Pointers can be compared using the relational operators (<, <=, >, >=) and the equality operators (== and !=).
  - Using relational operators is meaningful only for pointers to elements of the same array.
- The outcome of the comparison depends on the relative positions of the two elements in the array.
- After the assignments

```
p = &a[5];
q = &a[1];
```

  the value of `p <= q` is 0 and the value of `p >= q` is 1.

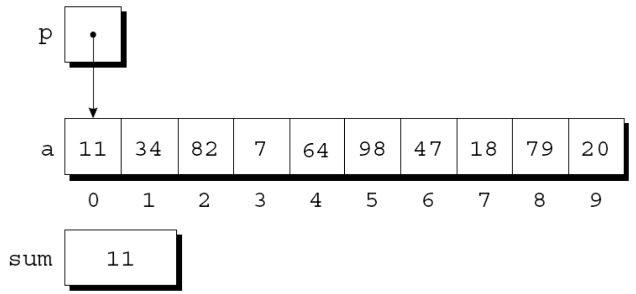
Using Pointers for Array Processing

- Pointer arithmetic allows us to visit the elements of an array by repeatedly incrementing a pointer variable.
- A loop that sums the elements of an array `a`:

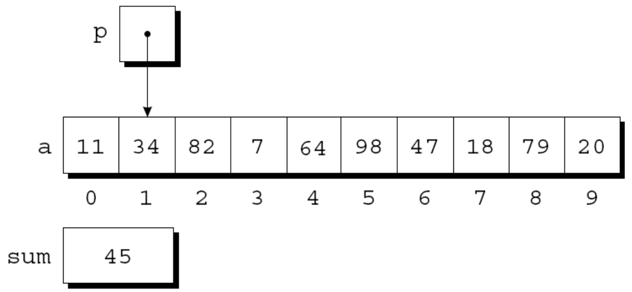
```
#define N 10
…
int a[N], sum, *p;
…
sum = 0;
for (p = &a[0]; p < &a[N]; p++)
   sum += *p;
```

Using Pointers for Array Processing

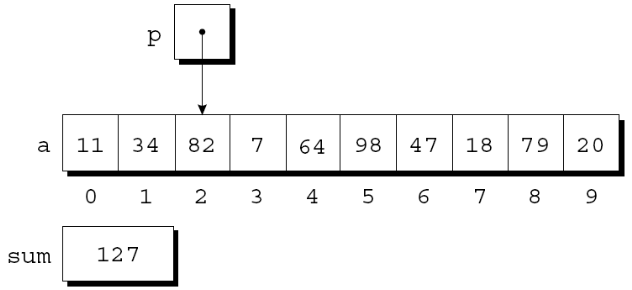
At the end of the first iteration:



At the end of the second iteration:



At the end of the third iteration:



Using Pointers for Array Processing

- The condition `p < &a[N]` in the `for` statement deserves special mention.
- It's legal to apply the address operator to `a[N]`, even though this element doesn't exist.
- Pointer arithmetic may save execution time.
- However, some C compilers produce better code for loops that rely on subscripting.

Using an Array Name as a Pointer

- Pointer arithmetic is one way in which arrays and pointers are related.
- Another key relationship:

  The name of an array can be used as a pointer to the first element in the array.

- This relationship simplifies pointer arithmetic and makes both arrays and pointers more versatile.

Using an Array Name as a Pointer

- Suppose that `a` is declared as follows:

```
int a[10];
```

- Examples of using `a` as a pointer:

```
*a = 7;     /* stores 7 in a[0] */
*(a+1) = 12;    /* stores 12 in a[1] */
```

- In general, `a + i` is the same as `&a[i]`.
  - Both represent a pointer to element `i` of `a`.
- Also, `*(a+i)` is equivalent to `a[i]`.
  - Both represent element `i` itself.

Using an Array Name as a Pointer

- The fact that an array name can serve as a pointer makes it easier to write loops that step through an array.
- Original loop:

```
for (p = &a[0]; p < &a[N]; p++)
  sum += *p;
```

- Simplified version:

```
for (p = a; p < a + N; p++)
  sum += *p;
```

Using an Array Name as a Pointer

UNIVERSITÀ DEGLI STUDI DI BRESCIA

- Although an array name can be used as a pointer, it's not possible to assign it a new value.
- Attempting to make it point elsewhere is an error:

```
while (*a != 0)
  a++;              /*** WRONG ***/
```

- This is no great loss; we can always copy `a` into a pointer variable, then change the pointer variable:

```
p = a;
while (*p != 0)
  p++;
```

reverse3.c – Pointer arithmetic

```c
/* Reverses a series of numbers (pointer version) */
#include <stdio.h>
#define N 10

int main(void){
    int a[N], *p;

    printf("Enter %d numbers: ", N);
    for (p = a; p < a + N; p++)
        scanf("%d", p);

    printf("In reverse order:");
    for (p = a + N - 1; p >= a; p--)
        printf(" %d", *p);
    printf("\n");

    return 0;
}
```

Dynamic Storage Allocation

- C's data structures, including arrays, are normally fixed in size.
  - Fixed-size data structures can be a problem, since we're forced to choose their sizes when writing a program.

- Fortunately, C supports **dynamic storage allocation**
- the ability to allocate storage during program execution.

- We can design data structures that grow (and shrink) as needed.

Memory Allocation Functions

UNIVERSITÀ DEGLI STUDI DI BRESCIA

- <stdlib.h> **header**

malloc —Allocates a block of memory but doesn’t initialize it.

calloc —Allocates a block of memory and clears it.

realloc —Resizes a previously allocated block of memory.

- These functions return a value of type void *
  - a “generic” pointer

Null Pointers

- If a memory allocation function can't locate a memory block of the requested size, it returns a **null pointer.**
  - A null pointer is a special value that can be distinguished from all valid pointers.

- After we've stored the function's return value in a pointer variable, we must test to see if it's a null pointer.

Null Pointers

- If a memory allocation function can't locate a memory block of the requested size, it returns a **null pointer.**
  - A null pointer is a special value that can be distinguished from all valid pointers.

- After we've stored the function's return value in a pointer variable, we must test to see if it's a null pointer.

- An example: testing malloc's return value:

```
if ((p = malloc(10000)) == NULL) {
    /* allocation failed; take appropriate action */
}
```

- NULL is a macro (defined in various library headers) that represents the null pointer.

Using malloc to Allocate Memory for a String

- Dynamic storage allocation is often useful for working with strings.
  - Strings are stored in character arrays → hard to anticipate they need to be.
  - By allocating strings dynamically, we can postpone the decision

- Prototype for the malloc function:

  ```
  void *malloc(size_t size);
  ```

- malloc allocates a block of size bytes and returns a pointer to it.

- size_t is an unsigned integer type defined in the library.

Using **malloc** to Allocate Memory for a String

- A call of malloc that allocates memory for a string of n characters:

```
p = malloc(n + 1);
```

- Each character requires one byte of memory; adding 1 to n leaves room for the null character.

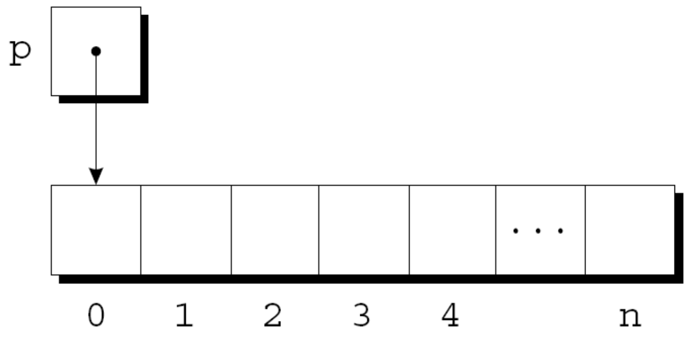
- Some programmers prefer to cast malloc’s return value, although the cast is not required:

```
p = (char *) malloc(n + 1);
```

  In this way p is a char * pointer

Using malloc to Allocate Memory for a String

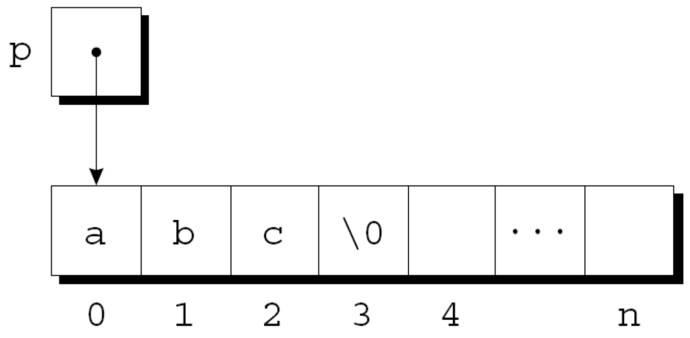
- Memory allocated using malloc isn’t cleared, so p will point to an uninitialized array of n + 1 characters:



- Calling strcpy is one way to initialize this array:

  strcpy(p, "abc");

  - The first four characters in the array will now be a, b, c, and \0:



Using Dynamic Storage Allocation in String Functions

A function that concatenates two strings without changing either one

```
char *concat(const char *s1, const char *s2){
    char *result;

    result = malloc(strlen(s1) + strlen(s2) + 1);
    if (result == NULL) {
        printf("Error: malloc failed in concat\n");
        exit(EXIT_FAILURE);
    }
    strcpy(result, s1);
    strcat(result, s2);
    return result;
}
```

Dynamically Allocated Arrays

- Dynamically allocated arrays have the same advantages as dynamically allocated strings.
- The close relationship between arrays and pointers makes a dynamically allocated array as easy to use as an ordinary array.
- Although malloc can allocate space for an array, the calloc function is sometimes used instead, since it initializes the memory that it allocates.
- The realloc function allows us to make an array “grow” or “shrink” as needed.

Using malloc to Allocate Storage for an Array

- Suppose a program needs an array of n integers, where n is computed during program execution.

- We'll first declare a pointer variable:

```
int *a;
```

- Once the value of n is known, the program can call malloc to allocate space for the array:

```
a = malloc(n * sizeof(int));
```

- Always use the sizeof operator to calculate the amount of space required for each element.

Using malloc to Allocate Storage for an Array

- We can now ignore the fact that a is a pointer and use it instead as an array name, thanks to the relationship between arrays and pointers in C.

- For example, we could use the following loop to initialize the array that a points to:

```
for (i = 0; i < n; i++)
  a[i] = 0;
```

- We also have the option of using pointer arithmetic instead of subscripting to access the elements of the array.

The calloc Function

- The calloc function is an alternative to malloc.

- Prototype for calloc:

```
void *calloc(size_t nmemb, size_t size);
```

- Properties of calloc:
  - Allocates space for an array with nmemb elements, each of which is size bytes long.
  - Returns a null pointer if the requested space isn’t available.
  - Initializes allocated memory by setting all bits to 0.

The realloc Function

- The realloc function can resize a dynamically allocated array.
- Prototype for realloc:

  ```
  void *realloc(void *ptr, size_t size);
  ```

- ptr must point to a memory block obtained by a previous call of malloc, calloc, or realloc.
- size represents the new size of the block, which may be larger or smaller than the original size.

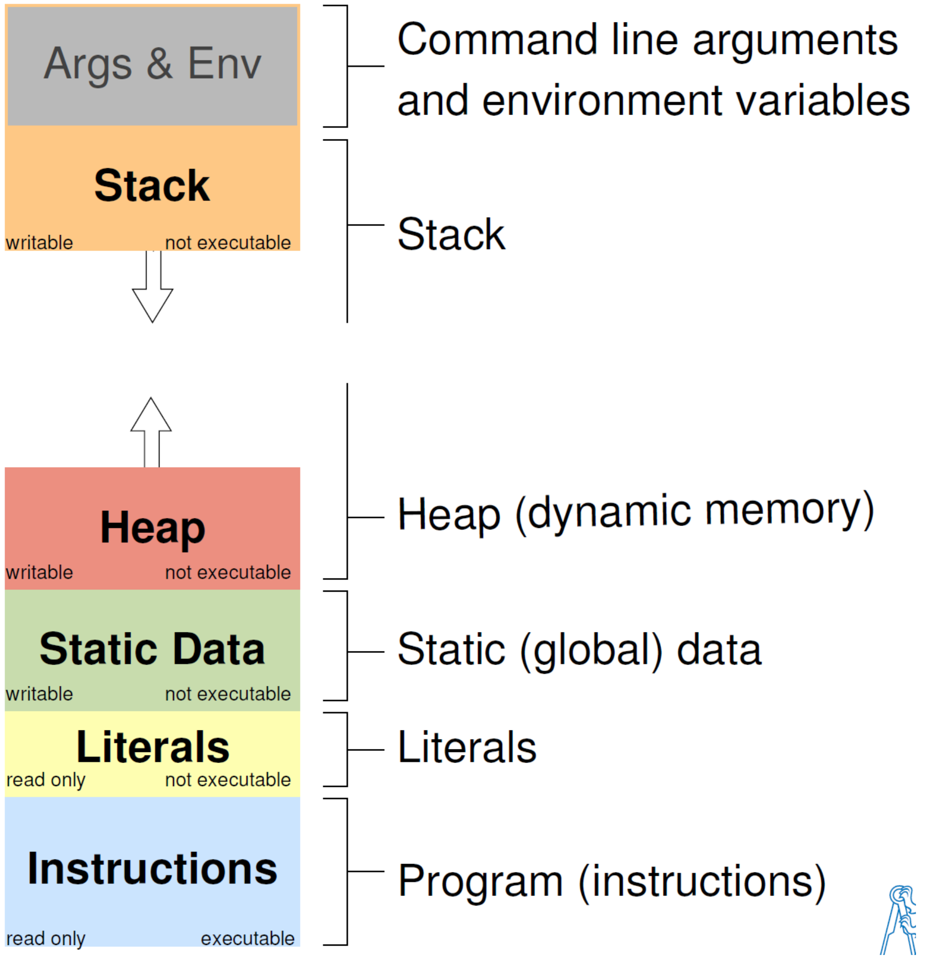
The realloc Function

- Properties of realloc:
  - When it expands a memory block, realloc doesn't initialize the bytes that are added to the block.
  - If for any reason realloc can't enlarge the memory block as requested, even after trying to reallocate it elsewhere → it can't do nothing than returning a null pointer
    - the data in the old memory block is unchanged.
  - If realloc is called with a null pointer as its first argument, it behaves like malloc
  - If realloc is called with 0 as its second argument, it frees the memory block

Deallocating Storage

UNIVERSITÀ DEGLI STUDI DI BRESCIA

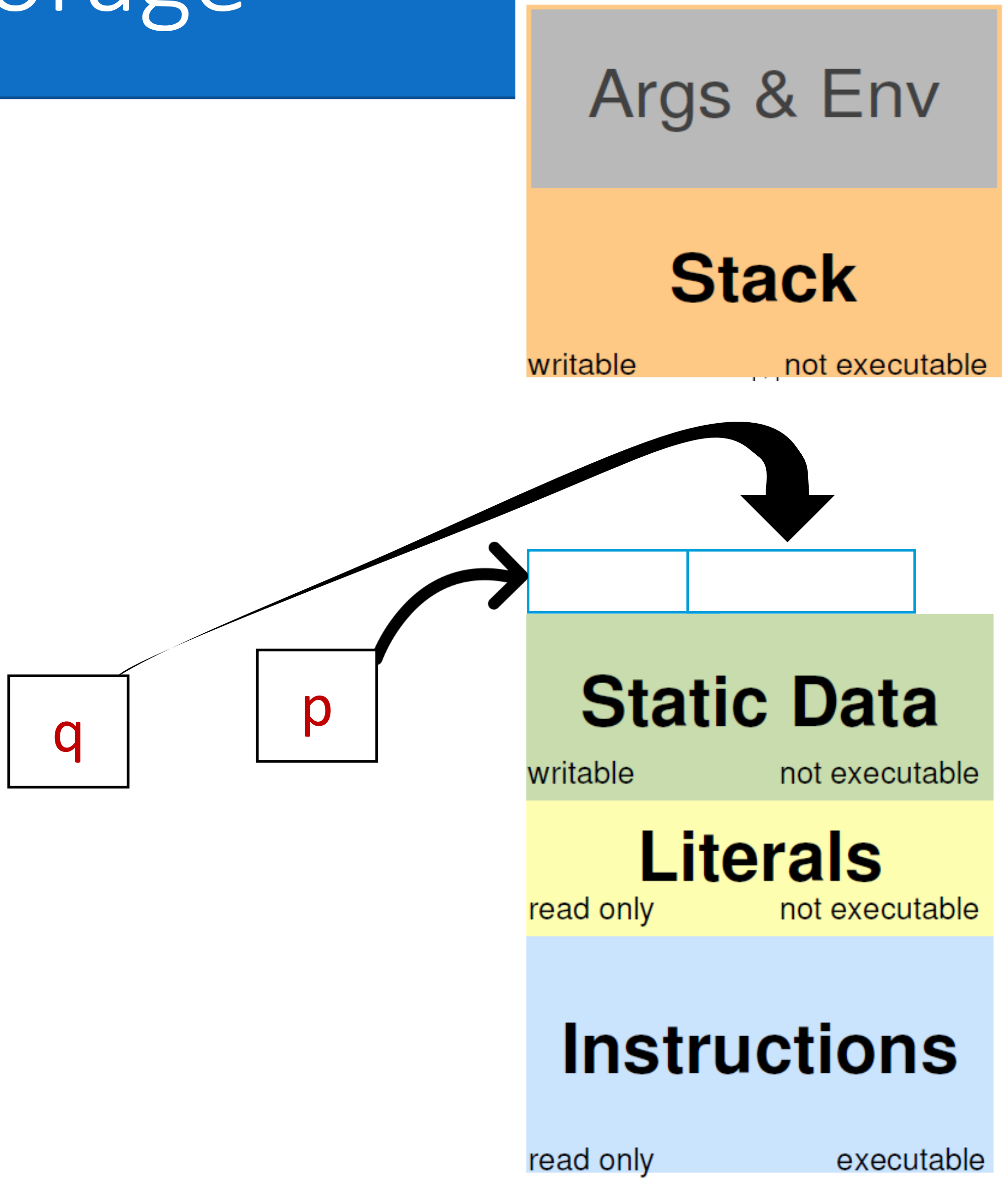
- malloc and the other memory allocation functions obtain memory blocks from a storage pool known as the **heap**.
- Calling these functions too often—or asking them for large blocks of memory—can exhaust the heap, causing the functions to return a null pointer.
- To make matters worse, a program may allocate blocks of memory and then lose track of them, thereby wasting space.



Deallocating Storage

- Example:

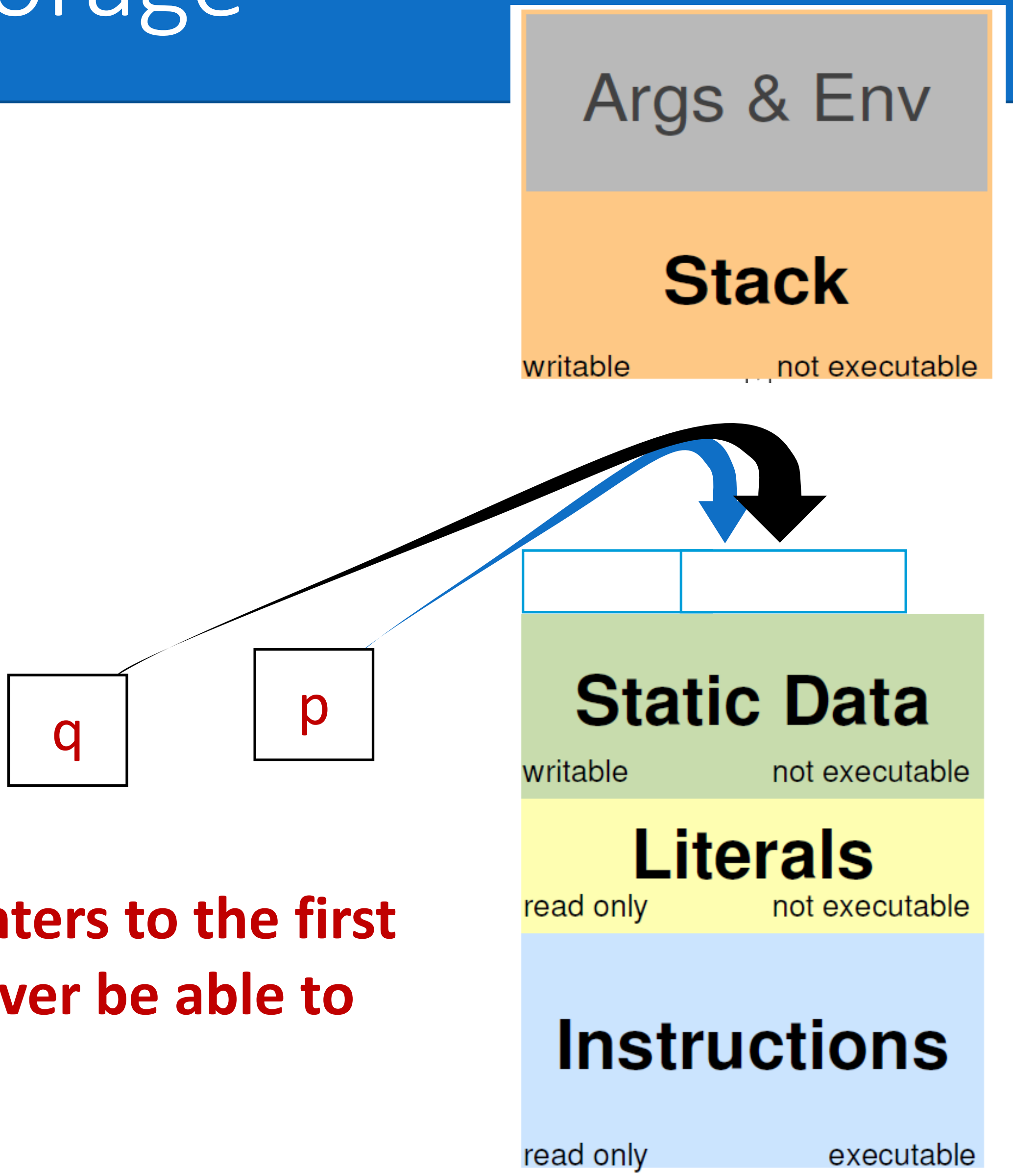
```
p = malloc(…);
q = malloc(…);
```



Deallocating Storage

- Example:

p = malloc(…);
q = malloc(…);
p = q;



There are no pointers to the first block, so we'll never be able to use it again.

Deallocating Storage

- A block of memory that's no longer accessible to a program is said to be **garbage**
- A program that leaves garbage behind has a **memory leak**

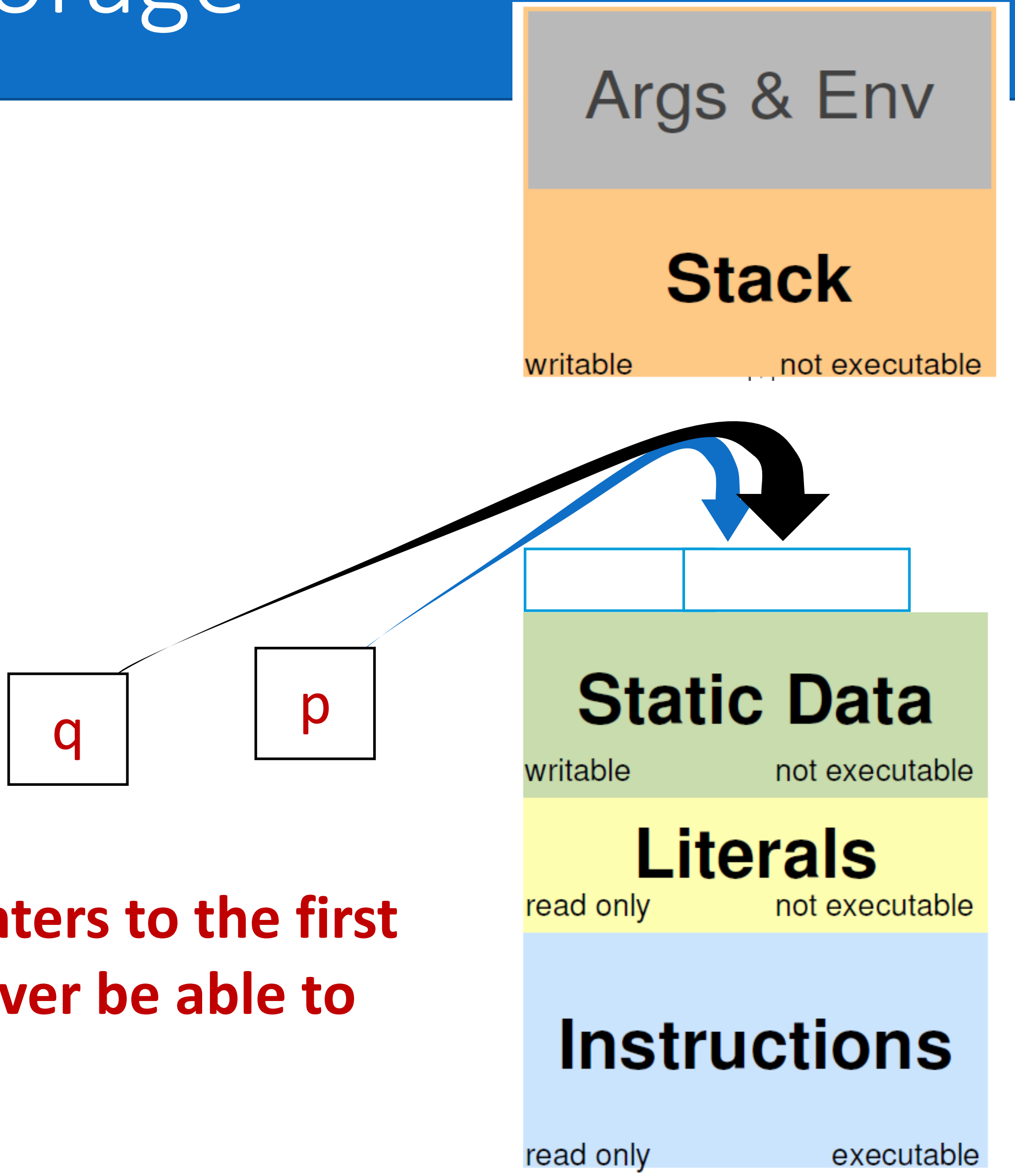
- Some languages provide a **garbage collector** that automatically locates and recycles garbage
    - Java
    - Python
    - **but C doesn't.**

- Instead, each C program is responsible for recycling its own garbage by calling the free function to release unneeded memory.

Deallocating Storage

- Example:

  p = malloc(…);
  q = malloc(…);
  p = q;



There are no pointers to the first block, so we'll never be able to use it again.

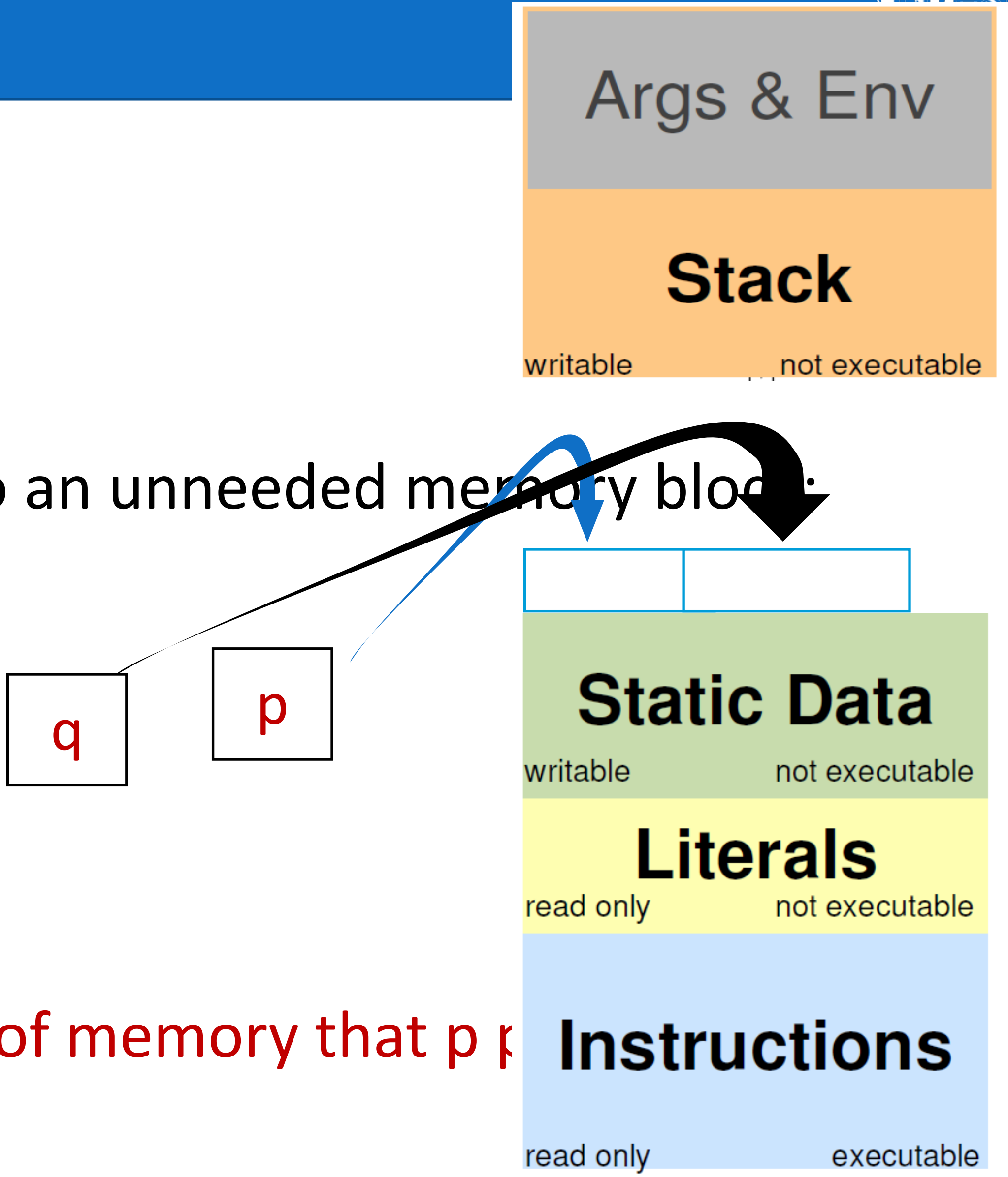
The free Function

- Prototype for free:

  void free(void *ptr);

- free will be passed a pointer to an unneeded memory block:

  p = malloc(…);
  q = malloc(…);
  free(p);
  p = q;



- Calling free releases the block of memory that p p

The "Dangling Pointer" Problem

UNIVERSITÀ DEGLI STUDI DI BRESCIA

- Using free leads to a new problem: **dangling pointers**

- free(p) deallocates the memory block that p points to
- but doesn't change p itself.

- If we forget that p no longer points to a valid memory block, chaos may ensue

```
char *p = malloc(4);
…
free(p);
…
strcpy(p, "abc");   /*** WRONG ***/
```

- Modifying the memory that p points to is a serious error.