

PROGRAMMAZIONE PROCEDURALE

A.A. 2025/2026



A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location.



DECLARING POINTERS



POINTERS

- ④ A *pointer* represents both the address and the type of an object. If an object or function has the type T , then a pointer to it has the derived type “*pointer to T*”.
- ④ For example, if **var** is a float variable, then the expression **&var**—whose value is the address of the float variable—has the type *pointer to float*, or in C notation, the type **float ***.
- ④ Because **var** doesn't move around in memory, the expression **&var** is a constant pointer.
- ④ The declaration of a pointer to an object that is not an array has the following syntax:

type * [type-qualifier-list] name [= initializer];

THE & OPERATOR

- ④ The *address operator* & yields the address of its operand. If the operand x has the type T , then the expression **& x** has the type “**pointer to T** ”
- ④ The operand of the address operator must have an addressable location in memory: the operand must designate either a function or an object (i.e., an *lvalue*) that is not a bit-field.
- ④ You need to obtain the addresses of objects and functions when you want to initialize pointers to them:

```
float x, *ptr;  
ptr = &x;           // OK: Make ptr point to x.  
ptr = &(x+1);      // Error: (x+1) is not an lvalue.
```

THE INDIRECTION OPERATOR *

- ④ Conversely, when you have a pointer and want to access the object it references, use the *indirection operator* *, which is sometimes called the *dereferencing operator*.
- ④ Its operand must have a pointer type.
- ④ If *ptr* is a pointer, then **ptr* designates the object or function that *ptr* points to.
- ④ If *ptr* is an object pointer, then **ptr* is an *lvalue*, and you can use it as the left operand of an assignment operator:

```
float x, *ptr = &x;  
*ptr = 1.7           // Assign the value 1.7 to the variable x  
++(*ptr);           // and add 1 to it.
```

INDIRECTION AND ARITHMETIC

- ④ Asterisk * with one operand is the **dereference** or **indirection operator**, and with two operands, it is the multiplication sign.
- ④ In each of these cases, the unary operator has higher precedence than the binary operator. For example, the expression ***ptr1 * *ptr2** is equivalent to **(*ptr1) * (*ptr2)**.
- ④ Look at the operator precedence/associativity table

QUESTION

@ Given

✓ `int *p;`

@ What is its type of p?

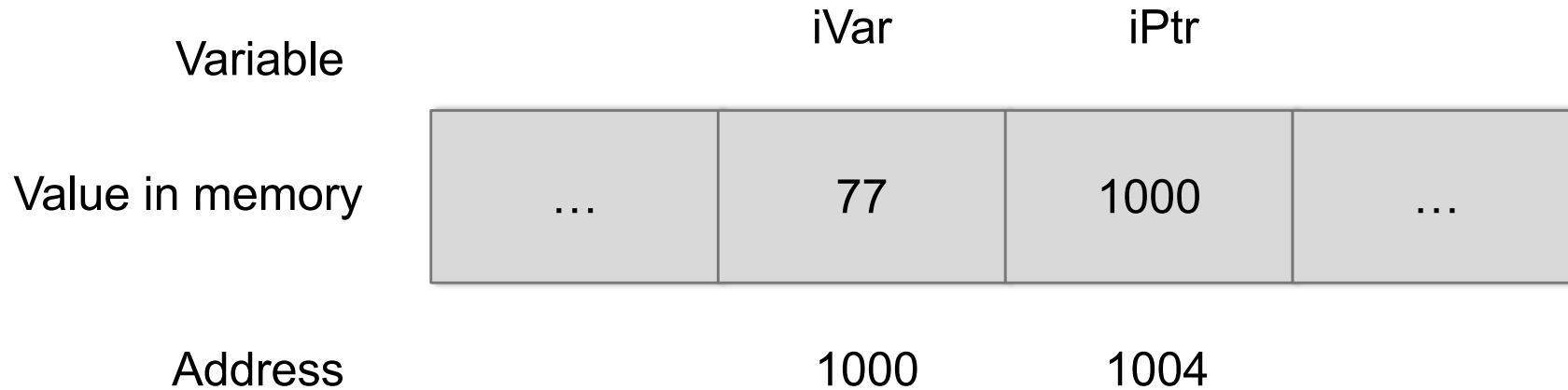
✓ `int?`

NO: its type is “pointer to int” or `int*`

IN MEMORY

© The addresses shown are purely fictitious examples.

```
int iVar = 77;  
int *iPtr = &iVar;
```



PRINT POINTERS

- ④ It is often useful to output addresses for verification and debugging purposes.
 - ✓ The printf() functions provide a format specifier for pointers: %p.

```
printf( "Value of iPtr (i.e. the address of iVar): %p\n"  
       "Address of iPtr:                %p\n", iPtr, &iPtr );
```

- ④ The size of a pointer in memory—given by the expression sizeof(iPtr), for example—is the same regardless of the type of object addressed.
- ④ 8 byte(?)

NULL POINTERS

- ⌚ A *null pointer constant* is an integer constant expression with the value 0.
- ⌚ The macro `NULL` is defined in *stdlib.h*.
- ⌚ A null pointer is always unequal to any valid pointer to an object or function.

EXAMPLE

@ Initialization

✓ `int *p = NULL;`

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

```
int main() {  
    int a= 3;  
    int* p= NULL;  
    *p= 6;  
}
```

Segmentation fault

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

```
int main() {  
    int a= 3;  
    int* p= &a;  
    *p= 6;  
}
```

VOID POINTERS

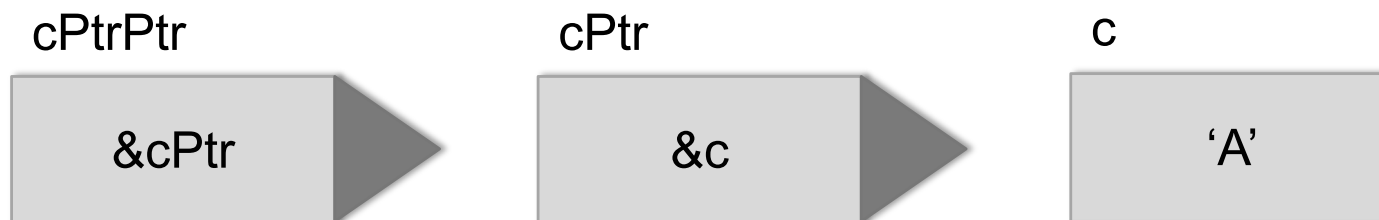
- ⌚ A pointer to void, or *void pointer* for short, is a pointer with the type `void *`.
- ⌚ As there are no objects with the type `void`, the type `void *` is used as the all-purpose pointer type.
 - ✓ A void pointer can represent the address of any object—but not its type.
- ⌚ To access an object in memory, you must always convert a void pointer into an appropriate object pointer.

```
void* pA= NULL;  
int p= 10;  
pA= &p;
```

```
printf("%d", *((int*) pA));
```

POINTERS TO POINTERS

- ⊙ A pointer variable is itself an object in memory, which means that a pointer can point to it.
- ⊙ To declare a pointer to a pointer, you must use two asterisks
 - ✓ `char c = 'A', *cPtr = &c, **cPtrPtr = &cPtr;`
- ⊙ The expression `*cPtrPtr` now yields the char pointer `cPtr`, and the value of `**cPtrPtr` is the char variable `c`.



EXAMPLE

The address of a is 0x7fff4fca4acc

```
int main(){
    int a= 2, *p= &a;
    printf("%d %d\n", *p, *&a);
    printf("%p %p\n", p, *&a);
}
```

```
MacBook-Francesco:Programmi francescosantini$ ./main
2 2
0x7fff4fca4acc 0x7fff4fca4acc
```

OPERATIONS WITH POINTERS



READ AND MODIFY

- ⌚ If **ptr** is a pointer, then ***ptr** designates the object (or function) that **ptr** points to.
- ⌚ The type of the pointer determines the type of object that is assumed to be at that location in memory.
- ⌚ For example, when you access a given location using an **int** pointer, you read or write an object of type **int**.

EXAMPLES

```
double x, y, *ptr;           // Two double variables and a pointer to double.
ptr = &x;                    // Let ptr point to x.
*ptr = 2.5;                  // Assign the value 2.5 to the variable x.
*ptr *= 2.0;                 // Multiply x by 2.
y = *ptr + 0.5;             // Assign y the result of the addition x + 0.5.
```

x is equal to 5.0
y is equal to 5.5

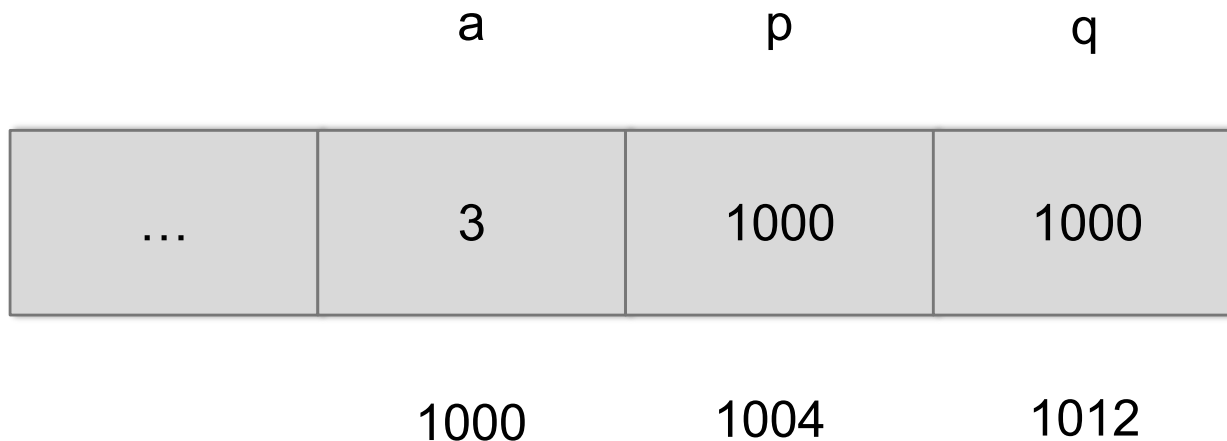
QUESTIONS

Ⓞ `int a= 3; int* p= &a;`

- ✓ Is “a” an lvalue? Yes
- ✓ Is “p” an lvalue? Yes
- ✓ Is “*p” an lvalue? Yes
- ✓ Is “&a” an lvalue? No

```
int a= 3;  
int* q= &a,  
int *p = q;
```

`&a == 1000`



OPERATIONS

- ④ The most important of these operations is accessing the object that the pointer refers to
- ④ You can also
 - ✓ compare pointers, and
 - ✓ use them to iterate through a memory block
- ④ Pointer arithmetics

POINTER ARITHMETICS

- ⌚ When you perform *pointer arithmetic*, the compiler automatically adapts the operation to the size of the objects referred to by the pointer type.
- ⌚ You can perform the following operations on pointers to objects:
 - ✓ Adding an integer to, or subtracting an integer from, a pointer.
 - ✓ Subtracting one pointer from another.
 - ✓ Comparing two pointers.

EXAMPLE ON COMPARING

```
int main() {
    int a= 5;
    int *p= &a;
    int *q= &a;

    if (p == q)
        printf("The two pointers are the same");

}
```

Comparison (== and !=) is used to check if two pointers point to the same location of memory

ARITHMETIC AND ARRAY OPERATIONS

- ⊙ The three pointer operations described here are generally useful only for pointers that refer to the elements of an array. To illustrate the effects of these operations, consider two pointers **p1** and **p2**, which point to elements of an array **a**:
 - ✓ If **p1** points to the array element **a[i]**, and **n** is an integer, then the expression **p2 = p1 + n** makes **p2** point to the array element **a[i+n]** (assuming that **i+n** is an index within the array **a**).
 - ✓ The subtraction **p2 - p1** yields the number of array elements between the two pointers, with the type **ptrdiff_t**. The type **ptrdiff_t** is defined in the header file *stddef.h*, usually as **int**. After the assignment **p2 = p1 + n**, the expression **p2 - p1** yields the value of **n**.
 - ✓ The comparison **p1 < p2** yields **true** if the element referenced by **p2** has a greater index than the element referenced by **p1**. Otherwise, the comparison yields false.

EXAMPLE

```
// Initialize an array and a pointer to its first element.
```

```
int dArr[5] = { 2, 1, 6, 3, 4 };
```

```
int *dPtr = dArr;
```

```
int i = 0;
```

```
dPtr = dPtr + 1;
```

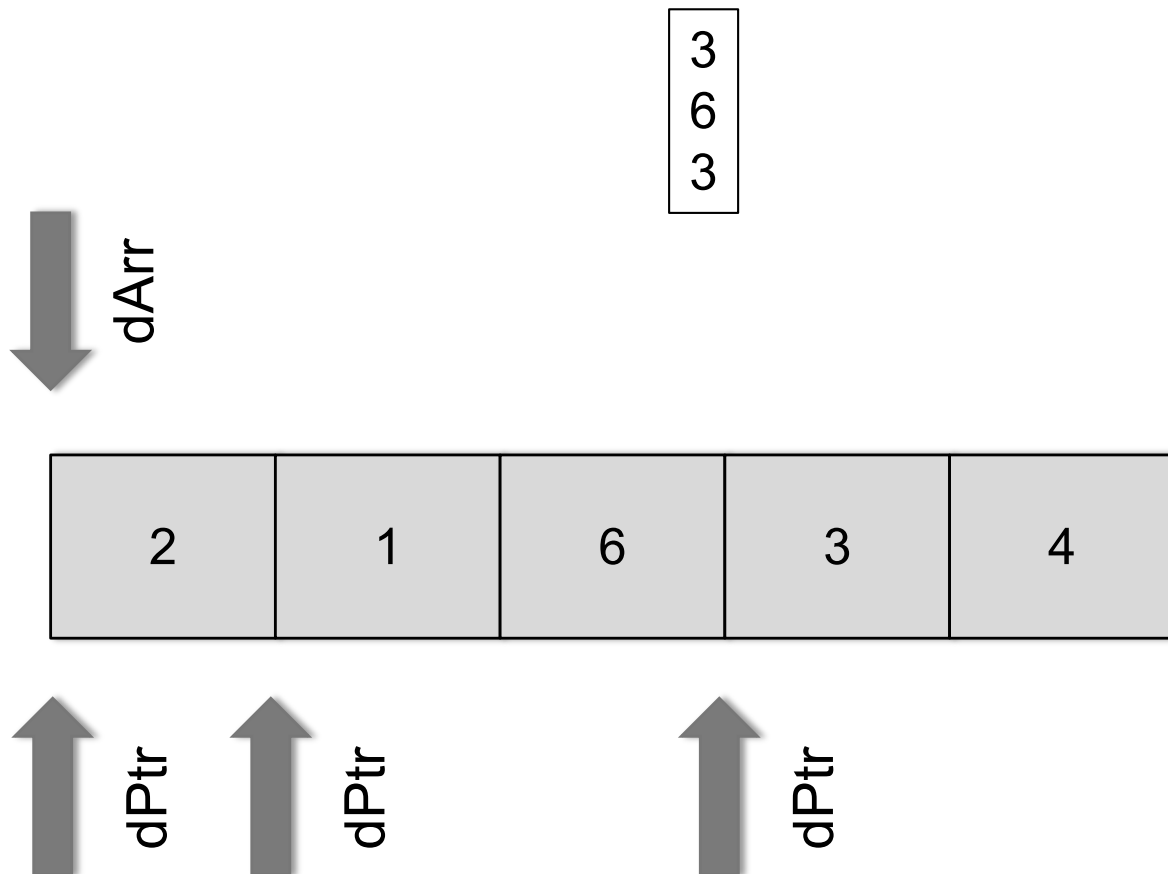
```
dPtr = 2 + dPtr;
```

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

```
printf( "%d\n", *(dPtr -1) );
```

```
i = dPtr - dArr;
```

```
printf( "%d\n", i );
```



CONSIDERATIONS ON THE EXAMPLE

- ⌚ The statement `dPtr = dPtr + 1;` adds the size of one array element to the pointer, so that `dPtr` points to the next array element, `dArr[1]`.
- ⌚ Because `dPtr` is declared as a pointer to `int`, its value is increased by `sizeof(int)`.
- ⌚ Subtracting one pointer from another yields an integer value with the type `ptrdiff_t`. The value is the number of objects that fit between the two pointer values.
- ✓ The type `ptrdiff_t` is defined in the header file `stddef.h`, usually as `int`.

MORE ON ARRAYS

- ⊙ Because the name of an array is implicitly converted into a pointer to the first array element wherever necessary, you can also substitute pointer arithmetic for array subscript notation:
 - ✓ The expression $a + i$ is a pointer to $a[i]$, and the value of $*(a+i)$ is the element $a[i]$.
 - ✓ **Arrays “do not exist in C”: they are just pointers**

L VALUES AND POINTERS

- ⊗ The operators that yield an lvalue include the subscript operator [] and the indirection operator *

Expression	Lvalue?
array[1]	
&array[1]	
ptr	
*ptr	
ptr+1	
*ptr+1	

int* ptr...

ONE MORE EXAMPLE

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

```
int main()
```

```
{
```

```
    // Initialize an array and a pointer to its first element.
```

```
    int dArr[5] = { 2, 1, 6, 3, 4 }, *dPtr = dArr;
```

```
    int i = 0;
```

```
    dPtr = dPtr + 1;
```

```
    printf("dArr %p\n", dArr);
```

```
    printf("dPtr %p\n", dPtr);
```

```
    dPtr = 2 + dPtr;
```

```
    printf("dPtr %p\n", dPtr);
```

```
}
```

dArr 0x7fff56845b19

dPtr 0x7fff56845b1d

dPtr 0x7fff56845b25

AN ADVANCED EXAMPLE

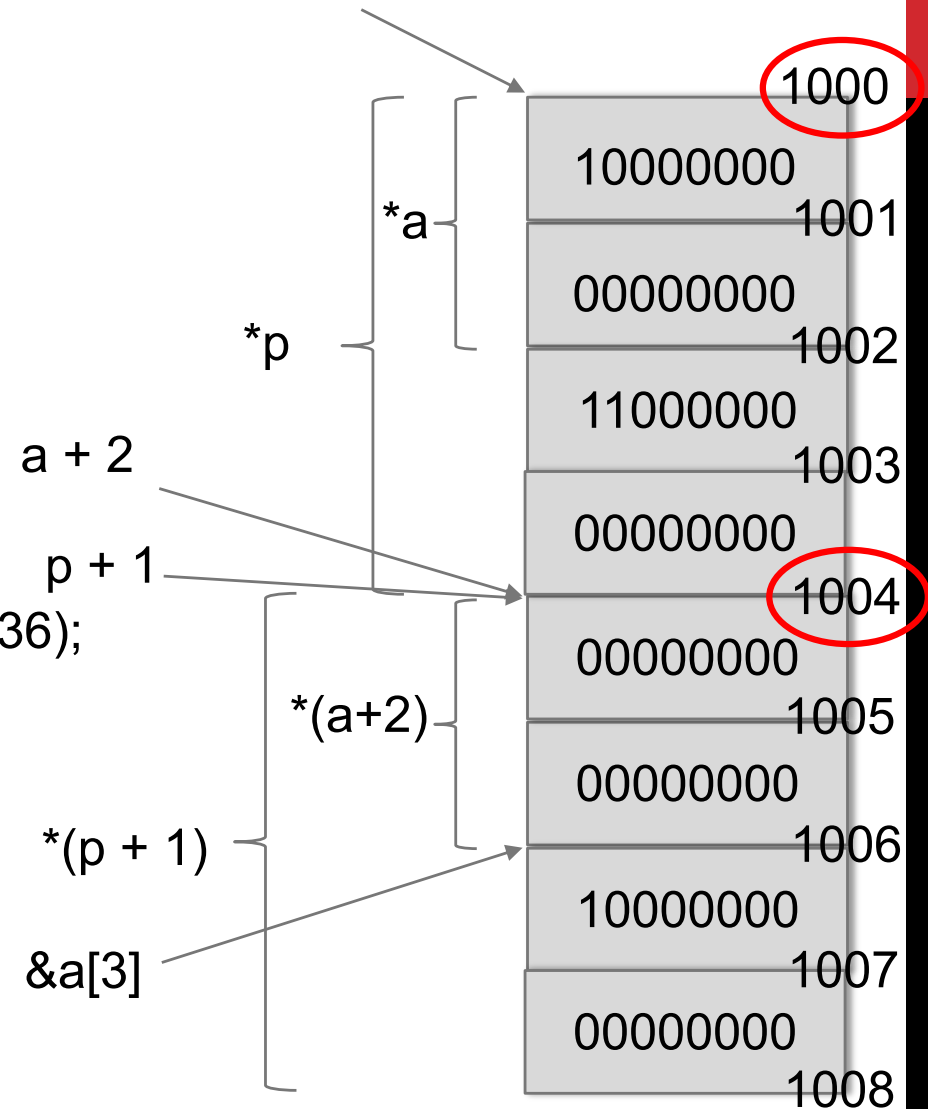
Little endian

*a == a[0]

p == a == &a[0]

```
int main() {
    short int a[4]= {1,3,[3]=1};
    int *p = (int*) a;

    printf("*a is equal to %d\n", *a);
    printf("*p==0 %d\n", *p== 0);
    printf("p == a %d\n", p == a);
    printf("**a+2 == 0 %d\n", *(a+2) == 0);
    printf("**p+1 == 65536 %d\n", *(p+1) == 65536);
    printf("&a[3] > (p + 1) %d\n", &a[3] > p+1);
    printf("%ld\n", (a+2) - &a[0]);
    printf("%d\n", ((int) (a+2)) - (int) (&a[0]) );
}
```



A short int is 2 bytes, an int is 4 bytes

$$1 \times 2^{16} + 1 \times 2^{17} + 1 \times 2^0 = 196609$$

EXAMPLE

```
int main() {
    short int a[4]= {1,3,[3]=1};
    int *p = (int*) a;

    printf("*a is equal to %d\n", *a);
    printf("*p == 0 %d\n", *p== 0);
    printf("p == a %d\n", p == a);
    printf("(a+2)= %d\n", *(a+2) == 0);
    printf("*p == 65536 %d\n", *(p+1) == 65536);
    printf("&a[3] > (p+1) %d\n", &a[3]> (p+1));
    printf("%ld\n", (a+2) - &a[0]);
    printf("%d\n", ((int) (a+2)) - (int) (&a[0]) );
}
```

```
MacBook-Francesco:esercizi francescosantini$ ./main
```

```
*a is equal to 1
```

```
*p == 0 0
```

```
p == a 1
```

```
*(a+2)== 0 1
```

```
*p == 65536 1
```

```
&a[3] > (p+1) 1
```

```
2
```

```
4
```

CONST POINTERS AND POINTERS TO CONST



CONSTANT POINTERS AND POINTERS TO CONSTANT VARS

- ⌚ It is possible to also define constant pointers.
- ⌚ When you define a constant pointer, you must also initialize it, because you can't modify it later.

```
int var, var2;           // Two objects with type int.
int *const c_ptr = &var; // A constant pointer to int.
*c_ptr = 123;           // OK: we can modify the object referenced, but ...
c_ptr = &var2;         // error: we can't modify the pointer.
```

POINTERS TO CONST

- ④ You can modify a pointer that points to an object that has a const-qualified type (also called a *pointer to const*).
- ④ However, you can use such a pointer only to read the referenced object, not to modify it
 - ✓ For this reason, pointers to const are commonly called “read-only pointers.”
- ④ You can use them if you want to be sure to not modify a variable through its pointer

EXAMPLE

```
int var; // An object with type int.

const int c_var = 100; // A constant int object.
const int *ptr_to_const; // A pointer to const int:
// the pointer itself is not constant!

ptr_to_const = &c_var; // OK: Let ptr_to_const point to c_var.

var = 2 * *ptr_to_const; // OK. Equivalent to: var = 2 * c_var;

ptr_to_const = &var; // OK: Let ptr_to_const point to var.

if ( c_var < *ptr_to_const ) // OK: "read-only" access.
    *ptr_to_const = 77; // Error: we can't modify var using
// ptr_to_const, even though var is
// not constant.
```

The assignment `ptr_to_const = &var` entails an implicit conversion: the int pointer value `&var` is automatically converted to the left operand's type, pointer to const int.

ONE MORE EXAMPLE

- 🕒 If you want to convert a pointer into a pointer to a less-qualified type, you must use an explicit type conversion.

```
int var;  
const int c_var = 100, *ptr_to_const;
```

```
int *ptr = &var;           // An int pointer that points to var.  
*ptr = 77;                // OK: ptr is not a read-only pointer.  
ptr_to_const = ptr;      // OK: implicitly converts ptr from "pointer to int"  
                          // into "pointer to const int".
```

```
*ptr_to_const = 77;      // Error: can't modify a variable through a read-only  
                          // pointer.
```

```
ptr = &c_var;             // Error: can't implicitly convert "pointer to const  
                          // int" into "pointer to int".
```

```
ptr = (int *) &c_var;    // OK: Explicit pointer conversions are always  
                          // possible.
```

SU LIBRO

© Sezioni 7.1-7.3, 7.5, 7.8, 7.9, 7.10