

# Pointers, References and Functions Parameters

30/09/2024

---

Danilo Ardagna, Federica Filippini

Politecnico di Milano  
[name.lastname@polimi.it](mailto:name.lastname@polimi.it)



POLITECNICO  
DI MILANO

# Pointers

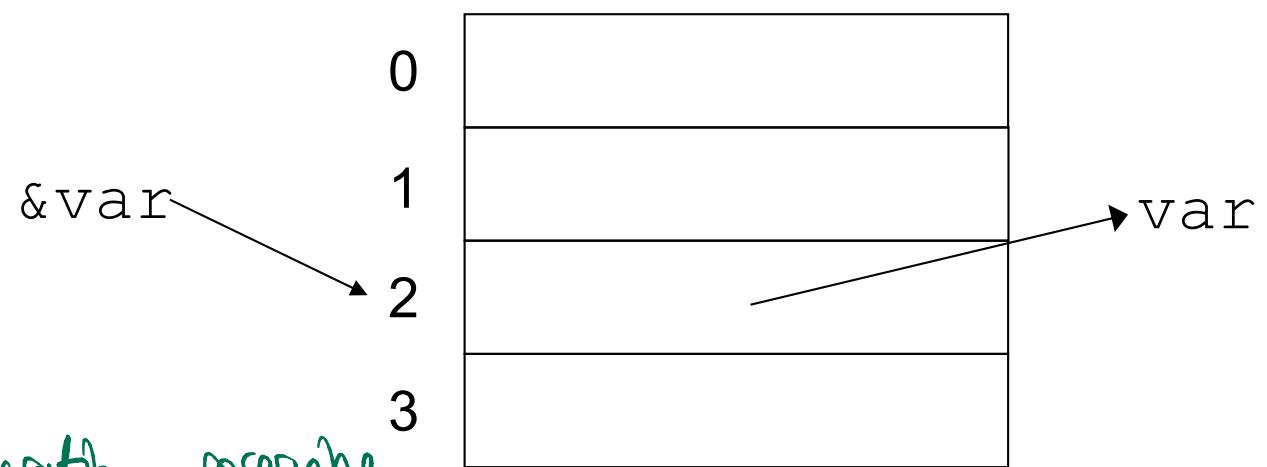
- Declaring a variable means reserving a memory area including several locations
- The number of locations depends on the type of data (e.g., integer 2 bytes, float 4 bytes, ...)
- Each memory location has a physical address and:
  - The name of the variable indicates the contents of the memory location
  - The operator & allows obtaining the memory address of the location associated with the variable to which the operator is applied

of  
next  
slide



# Variables and Memory

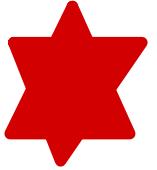
```
int var;
```



*declaration = reserving  
↑ a memory address.*

- Suppose the declaration reserves the memory area at address 2
- var indicates the content of the memory location
- &var indicates the memory address

*“&” = indicating the memory address.*



# Pointers

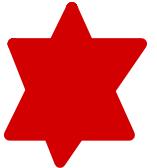
- Pointer variables store memory addresses
  - A pointer `p` can hold the address of a memory location
  - Think of them as a kind of integer values
- When declaring a pointer you must also specify what type of object the pointer points to

- Syntax:

```
double *p;
```

*"type \* name\_of\_pointer"*

- A pointer variable usually requires 2 bytes or 4 bytes depending on the architecture



# Dereferencing a pointer

- You can apply the dereferencing `*` operator to a pointer variable



- `*p` indicates the content of the location pointed by `p`

- If `p` is a pointer to an integer then `*p` is a simple integer

) Example

```
int * p; // declaration of the pointer  
// some code
```

`*p = 5;` // OK. `*p` is an integer

`p = 5;` // error, `p` is a pointer , whereas 5 is an int

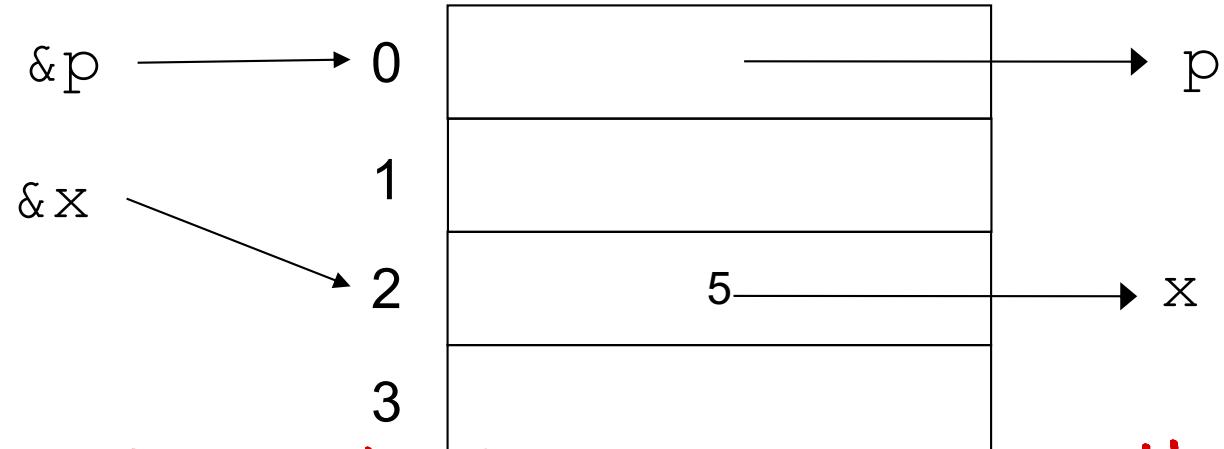
- Warning! The symbol `*` is used both in the declaration and in the dereferencing

# Operations

- We can store in a pointer variable the address of another variable

```
int x;
int *p;

x=5;
p=&x; // *p is 5
```

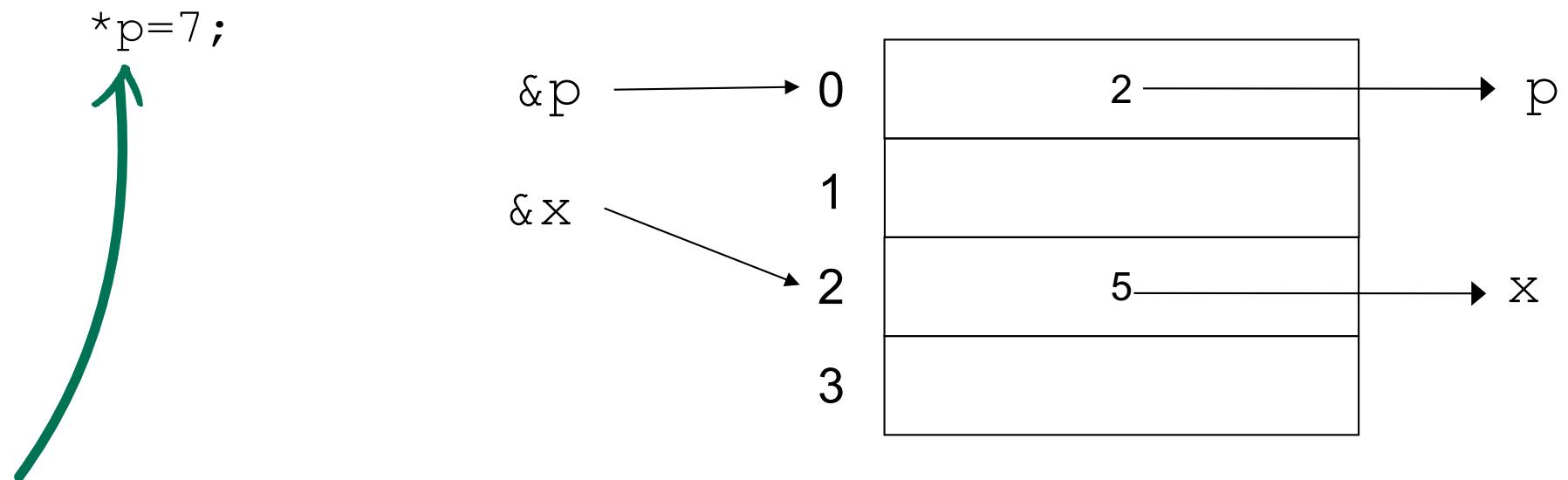


- p will point to the memory area where the value of x is stored
- A pointer's type determines how the memory referred to by the pointer's value is used
  - e.g., what an `int*` points to can be added but not, say, concatenated

*remember: &var indicates the memory address of the variable var.*

# Operations

- What happens if we perform the assignment?

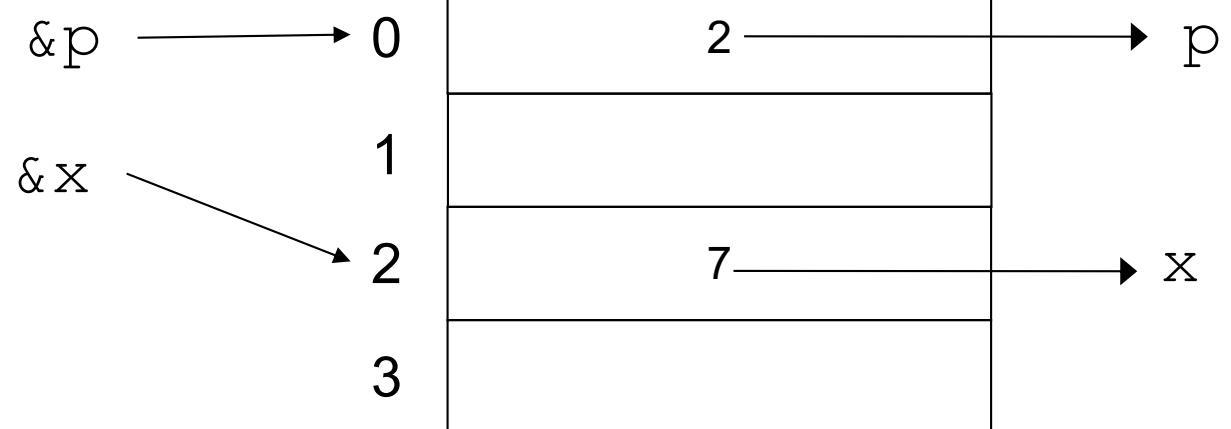


cf last slide, p is a pointer which points @ the  
memory address &x.

# Operations

- What happens if we perform the assignment?

`*p=7; // x get 7!`



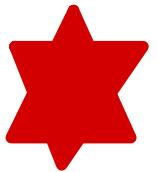
given what we just wrote (cf previous slide),  
x gets 7.

# Function Parameters

---

# Recap: Why functions?

- Chop a program into manageable pieces
  - “divide and conquer”
- Match our understanding of the problem domain
  - Name logical operations
  - A function should do one thing well
- Functions make the program easier to read
- A function can be useful in many places in a program
- Ease testing, distribution of labor, and maintenance
- Keep functions small
  - Easier to understand, specify, and debug



# Functions

- General form:

- return\_type name (formal arguments); // a declaration
- return\_type name (formal arguments) {body} // a definition
- For example:

```
double f(int a, double d);  
double f(int a, double d) { return a*d; }
```

- Formal arguments are often called parameters

- If you don't want to return a value give void as the return type

```
void increase_power_to(int level);
```

Here, void means “doesn't return a value”

- A body is a block
  - { /\* code \*/ } // a block

# Function Parameters

```
double circ(double radius) {  
    double res;  
    res = radius * 3.14 * 2;  
    radius = 7; // No sense instruction, let's see what happens to radius  
    return res;  
}  
  
// somewhere in the main  
double c;  
double r=5;  
  
c = circ(r);
```

# Function Parameters

```
double circ(double radius) {
    double res;
    res = radius * 3.14 * 2;
    radius = 7; // No sense instruction, let's see what happens to radius
    return res;
}
```

```
// somewhere in the main
double c;
double r=5;

c = circ(r);
```

*// r is still 5.0*

Formal parameter

- **Pass by value:**
  - Actual parameter copied within the formal parameter
- **Pass by reference**
  - Formal and actual parameter are the same object
  - We can return multiple results from functions!

Two ways of passing arguments.



# Function Parameters

```
double circ(double radius) {  
    double res;  
    res = radius * 3.14 * 2;  
    radius = 7; // No sense instruction, let's see what happens to radius  
    return res;  
}
```

Formal parameter

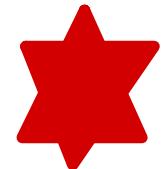
```
// somewhere in the main  
double c;  
double r=5;
```

```
c = circ(r);
```

Actual parameter



YOU HAVE TO DISTINGUISH BETWEEN "actual parameter"  
& "formal parameter"

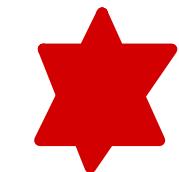


## Formal and Actual parameters

- In a function definition we use **formal parameters** representing a symbolic reference (identifiers) to objects used within the function
  - `radius` is a formal parameter
  - They are used by the function as if they were local variables
- The initial value of formal parameters is defined when the function is called using the **actual parameters** specified by the caller
  - `r` in our running example

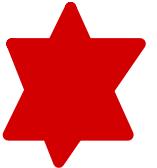


the one "with the true value",  
when you call the function.



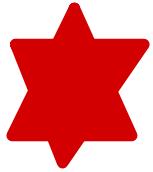
# Parameters Passing

- In a function call, parameters passing consists in associating the actual parameters with the formal parameters
- If the function prototype is:  
`double circ (double radius);`
- If we write:  
`c = circ (5.0);`
- `radius` (the formal parameter) will assume the value 5.0 (the actual parameter) for that particular invocation
- The exchange of information with the passing of parameters between caller and callee can take place in two ways:
  - Pass by value
  - Pass by reference



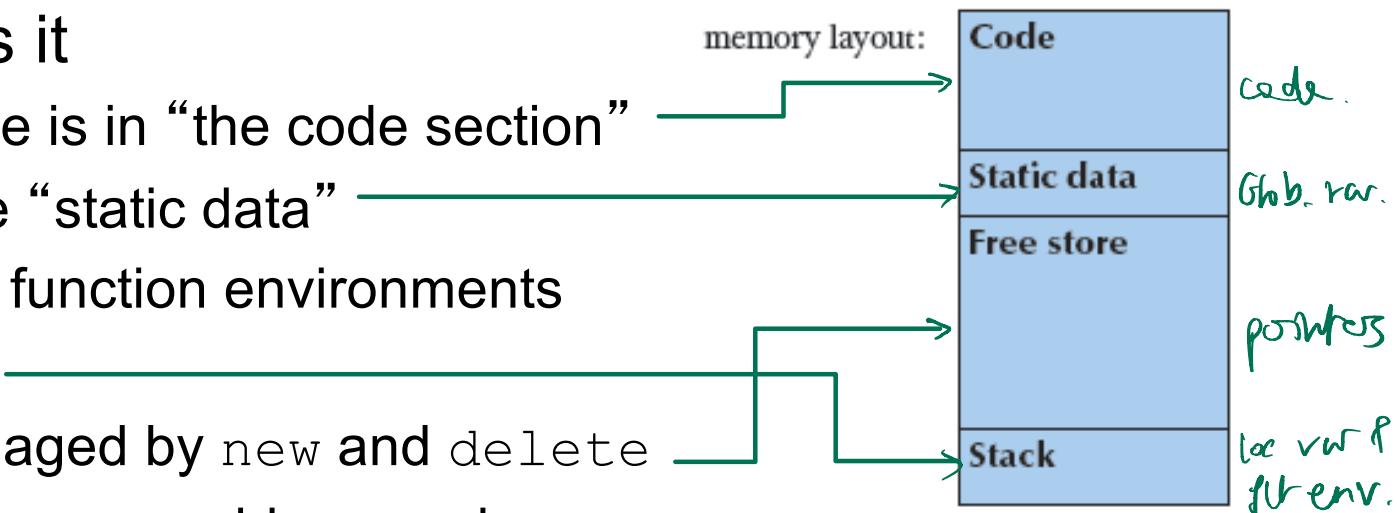
# Pass by value

- At the time of the function call, the value of the actual parameter is **copied** into the memory location of the corresponding formal parameter. In other words, the formal parameter and the actual parameter refer to **two different memory locations**
- The function works in its **environment** and therefore on **formal parameters**
- The **actual parameters** are not changed



# The computer's memory

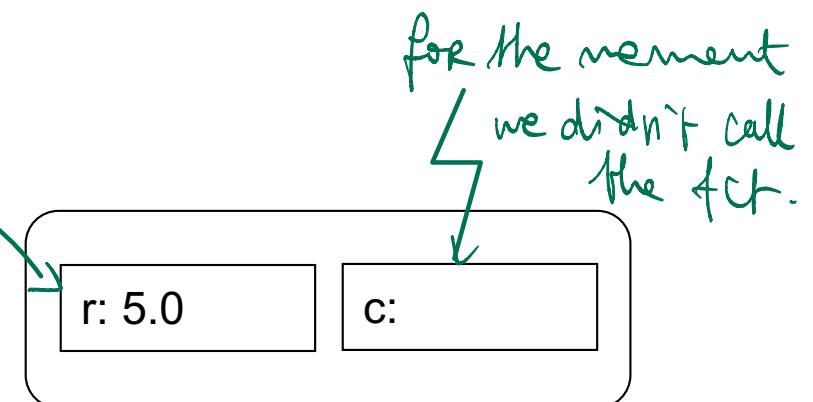
- As a program sees it
  - The executable code is in “the code section”
  - Global variables are “static data”
  - Local variables and function environments “live on the stack”
  - “Free store” is managed by `new` and `delete` (which work with memory addresses, i.e., are used with pointer variables)



# Pass by value example

```
double circ(double radius) {  
    double res;  
    res = radius * 3.14 *2;  
    radius = 7; /* No sense instruction, let's  
        see what happens to radius */  
    return res;  
}
```

```
// somewhere in the main  
double c;  
double r=5;  
  
c = circ(r);
```



- Functions environment are stored in the Stack
- The same mechanism is used to store block variables

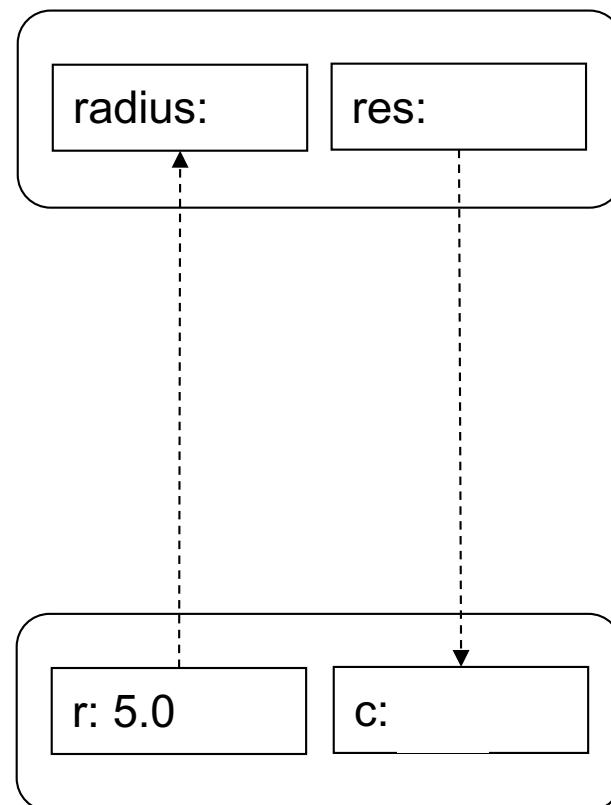
**main function environment**

# Pass by value example

```
double circ(double radius) {  
    double res;  
    res = radius * 3.14 *2;  
    radius = 7; /* No sense instruction, let's  
        see what happens to radius */  
    return res;  
}
```

```
// somewhere in the main  
double c;  
double r=5;  
  
c = circ(r);
```

**circ environment**



- Functions environment are stored in the Stack
- The same mechanism is used to store block variables

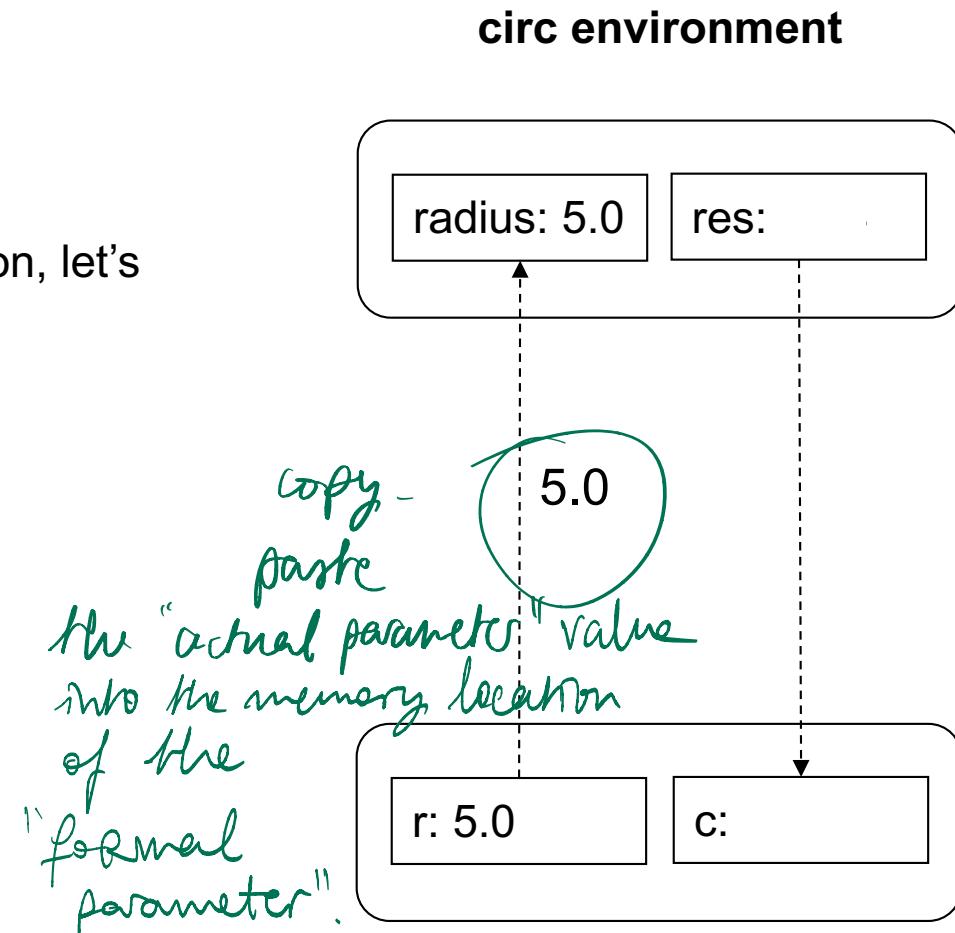
**main function environment**

# Pass by value example

```
double circ(double radius) {
    double res;
    res = radius * 3.14 * 2;
    radius = 7; /* No sense instruction, let's
        see what happens to radius */
    return res;
}

// somewhere in the main
double c;
double r=5;

c = circ(r);
```



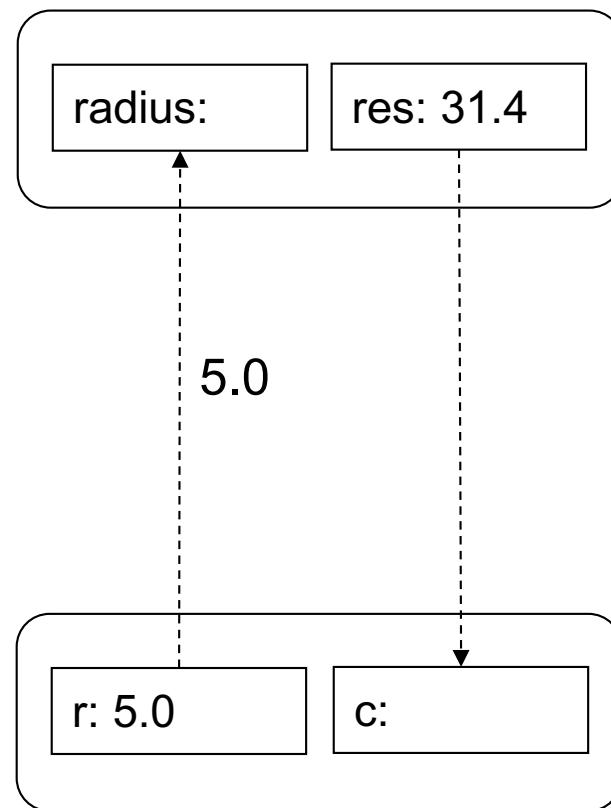
- Functions environment are stored in the Stack
- The same mechanism is used to store block variables

## main function environment

# Pass by value example

```
double circ(double radius) {  
    double res;  
    res = radius * 3.14 *2;  
    radius = 7; /* No sense instruction, let's  
        see what happens to radius */  
    return res;  
}  
  
// somewhere in the main  
double c;  
double r=5;  
  
c = circ(r);
```

**circ environment**



- Functions environment are stored in the Stack
- The same mechanism is used to store block variables

**main function environment**

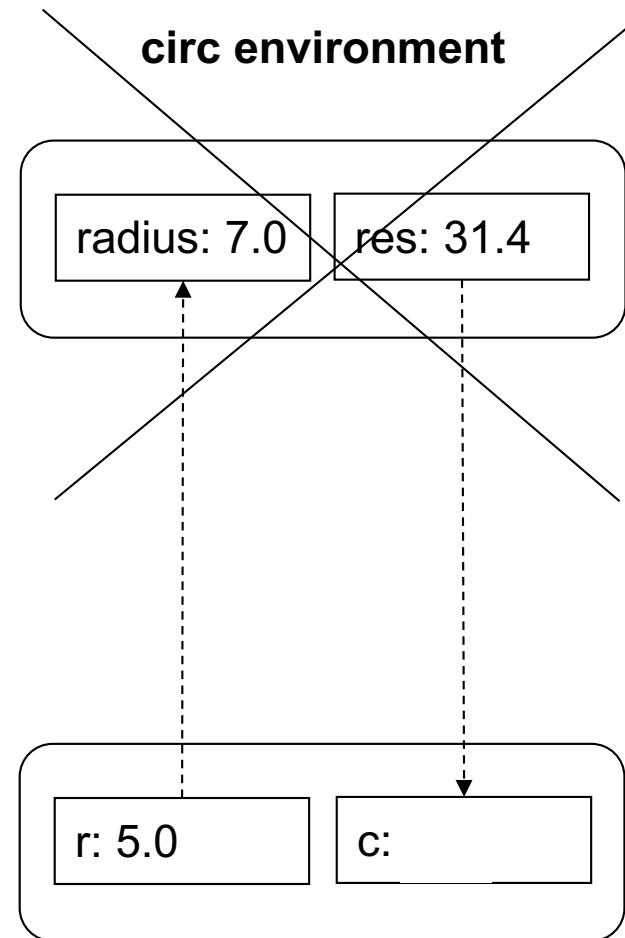
# Pass by value example

```
double circ(double radius) {
    double res;
    res = radius * 3.14 *2;
    radius = 7; /* No sense instruction, let's
                   see what happens to radius */
    return res;
}

// somewhere in the main
double c;
double r=5;

c = circ(r);
// r is still 5.0
```

- Functions environment are stored in the Stack
- The same mechanism is used to store block variables



**main function environment**



# Pass by reference



- At the time of the call the address of an actual parameters is associated with the formal parameters
- In other words, the actual parameter and the formal parameter share the same memory location ) This is where it's ≠ from pass by value.
- The running function works in its environment on the formal parameters (and consequently also on the actual parameters) and each change on the formal parameter is reflected to the corresponding actual parameter ) since they have the same memory loc.
- The function execution effects the caller with modifications to its the caller environment
  - In this way we can return multiple results!

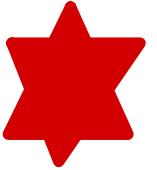
# Parameters passing in C

- In C there is no syntax mechanism to distinguish between parameters passing by value and by address
- Parameters passing is always by value:

```
double circ(double radius);  
/*pass by value*/
```

- To implement pass by reference we need to rely on pointers

```
double circ(double *radius );  
/*pass by reference*/
```



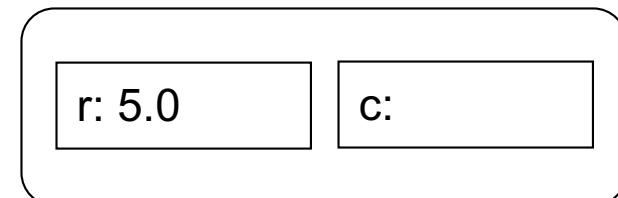
# Pass by reference



- We need:
  - A pointer for each formal parameter
  - The dereference operator in the function body to access the actual parameter
  - In the function call, the address of the actual parameter is used
- Warning! Arrays are always passed by reference
  - The name of an array variable, is an address, i.e., it is a pointer!
  - Why?
    - Efficiency!

# Pass by reference example

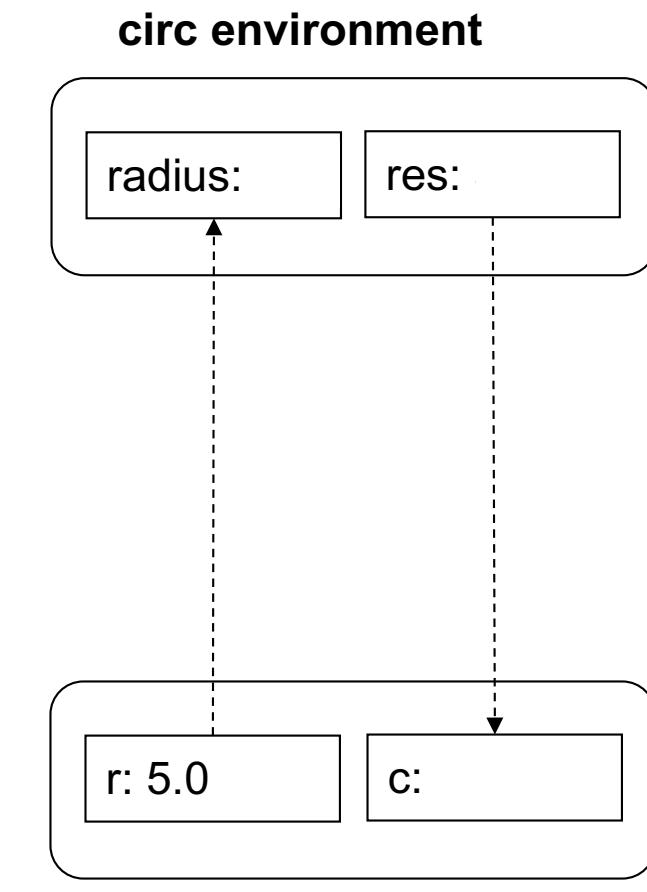
```
double circ(double *radius) {  
    double res;  
    res = *radius * 3.14 *2;  
    *radius = 7; /* No sense instruction, let's see  
        what happens to radius */  
    return res;  
}  
  
// somewhere in the main  
double c;  
double r = 5;  
  
c = circ(&r);
```



main function environment

# Pass by reference example

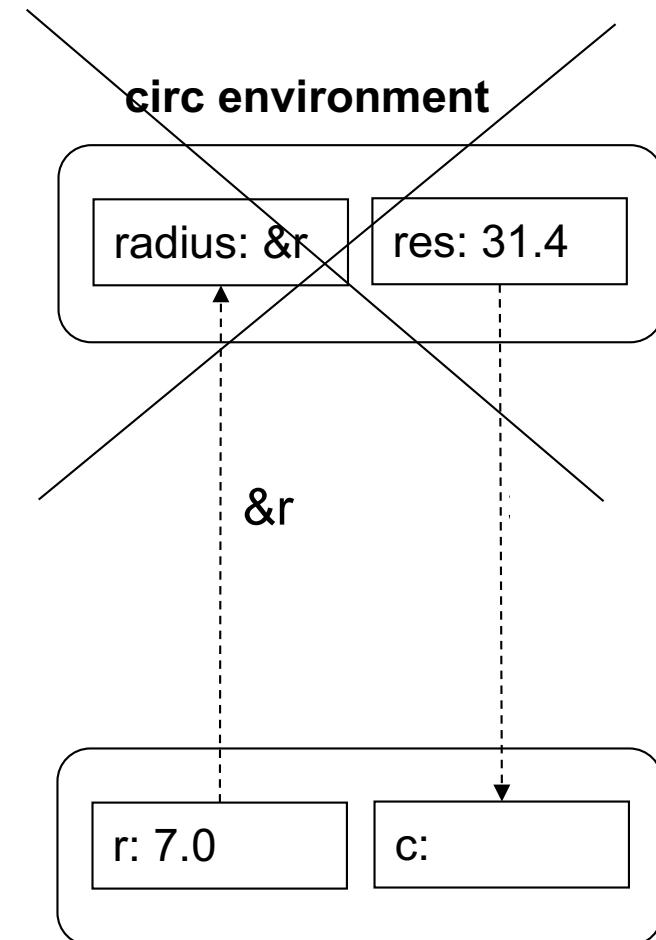
```
double circ(double *radius) {  
    double res;  
    res = *radius * 3.14 * 2;  
    *radius = 7; /* No sense instruction, let's see  
        what happens to radius */  
    return res;  
}  
  
// somewhere in the main  
double c;  
double r = 5;  
  
c = circ(&r);
```



# Pass by reference example

```
double circ(double *radius) {  
    double res;  
    res = *radius * 3.14 * 2;  
    *radius = 7; /* No sense instruction, let's see  
        what happens to radius */  
    return res;  
}  
  
// somewhere in the main  
double c;  
double r=5;  
  
c=circ(&r);  
//Warning! Now r is 7.0
```

What about circ(22.0);?

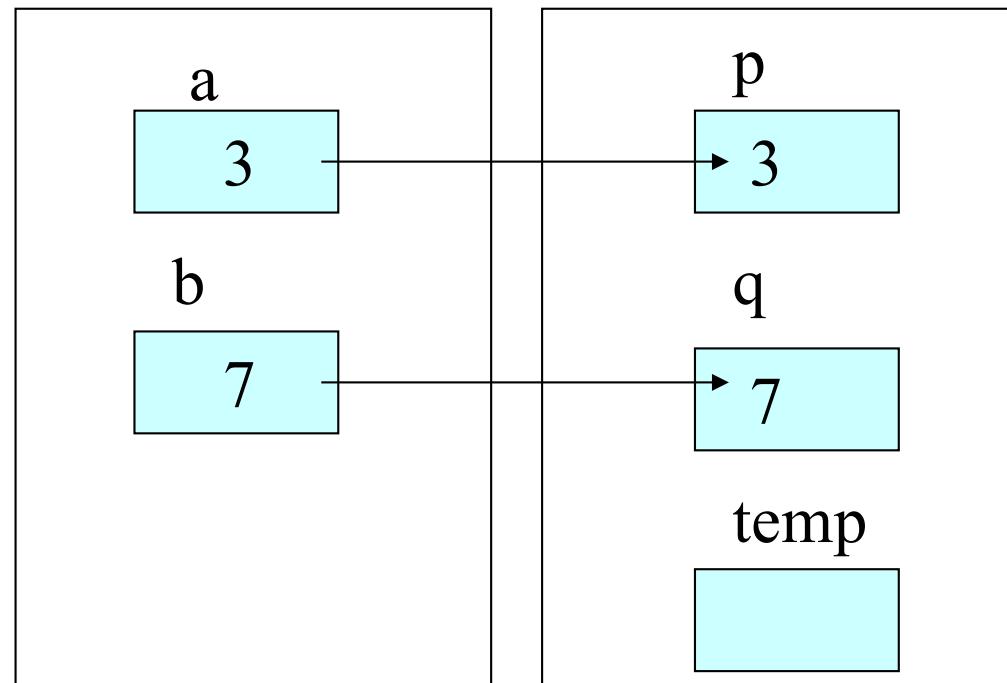


main function environment

# Example: swap of 2 ints

```
void swap (int p, int q) {  
    int temp;  
    temp = p;  
    p = q;  
    q = temp;  
}
```

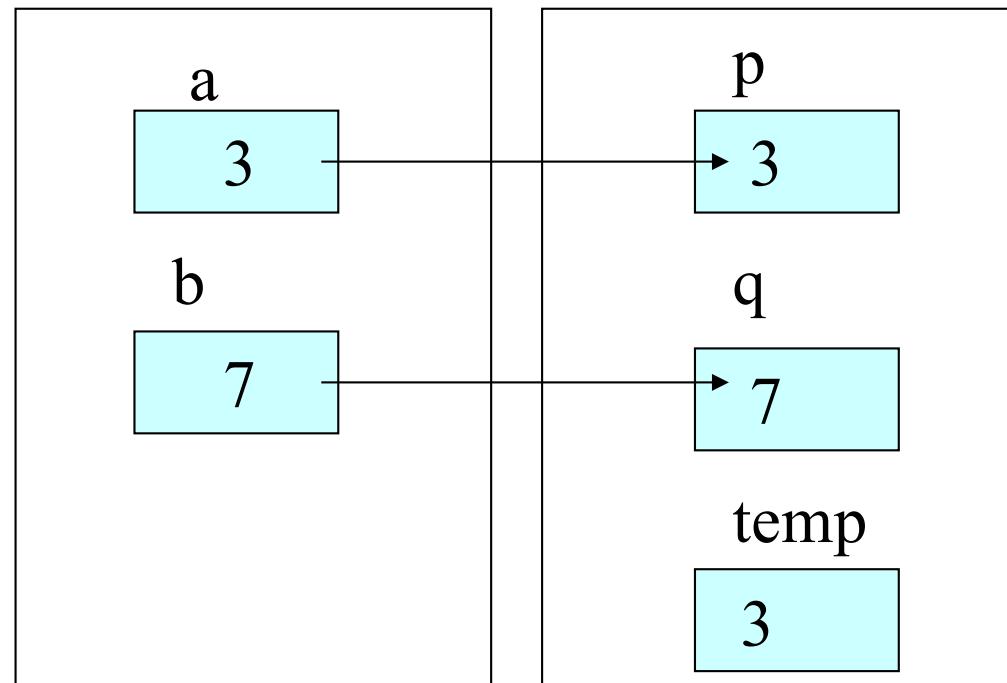
In main: swap (a, b)



# Example: swap of 2 ints

```
void swap (int p, int q) {  
    int temp;  
    temp = p;  
    p = q;  
    q = temp;  
}
```

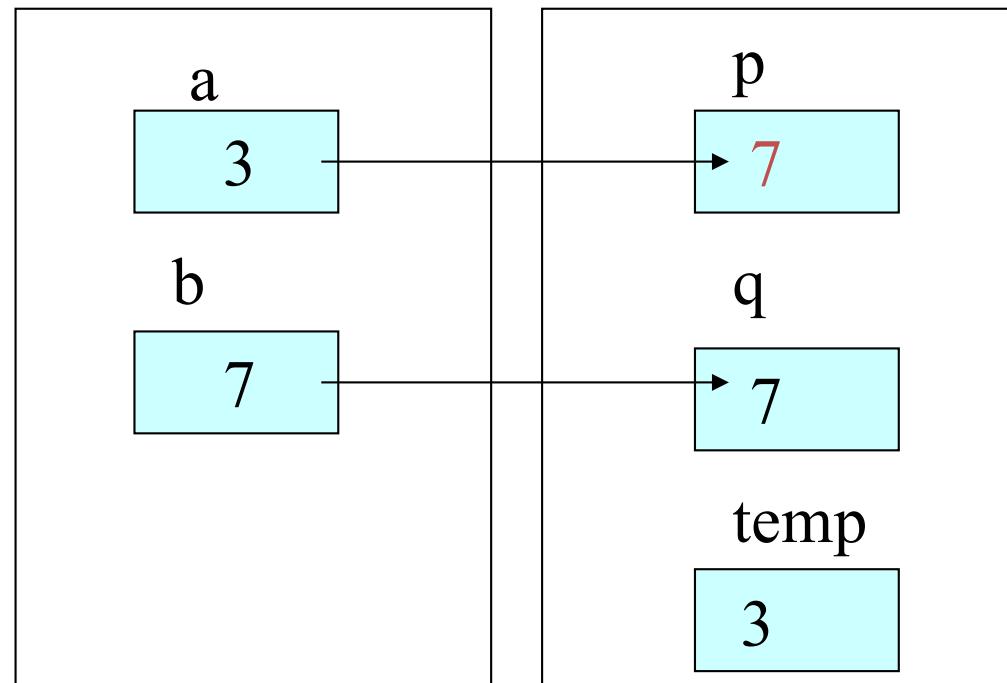
In main: swap (a, b)



# Example: swap of 2 ints

```
void swap (int p, int q) {  
    int temp;  
    temp = p;  
    → p = q;  
    q = temp;  
}
```

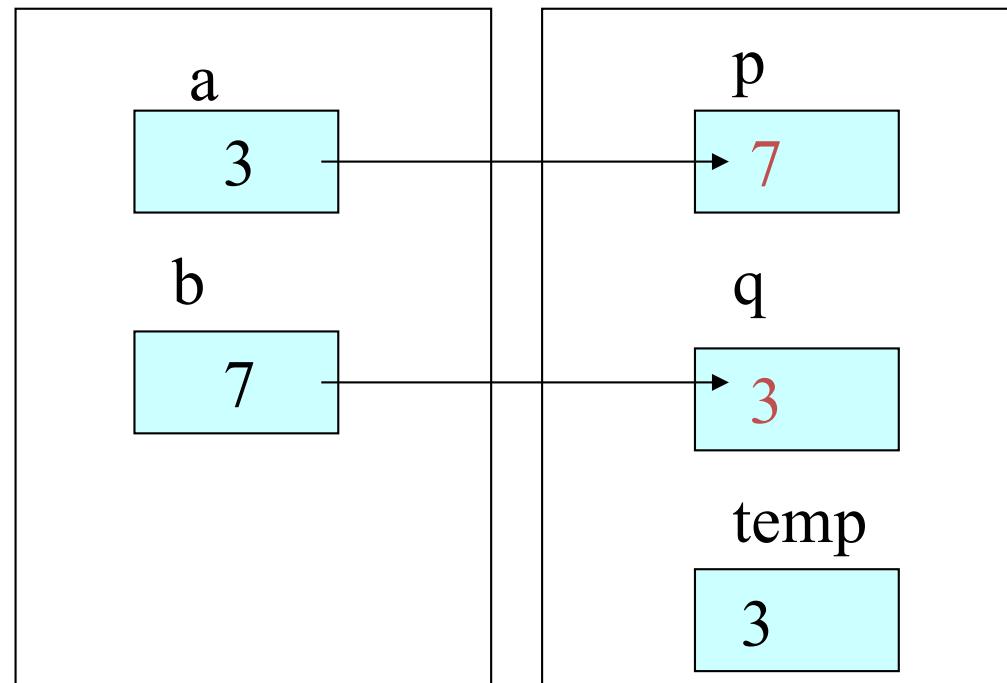
In main: swap (a, b)



# Example: swap of 2 ints

```
void swap (int p, int q) {  
    int temp;  
    temp = p;  
    p = q;  
    q = temp;  
}
```

In main: swap (a, b)

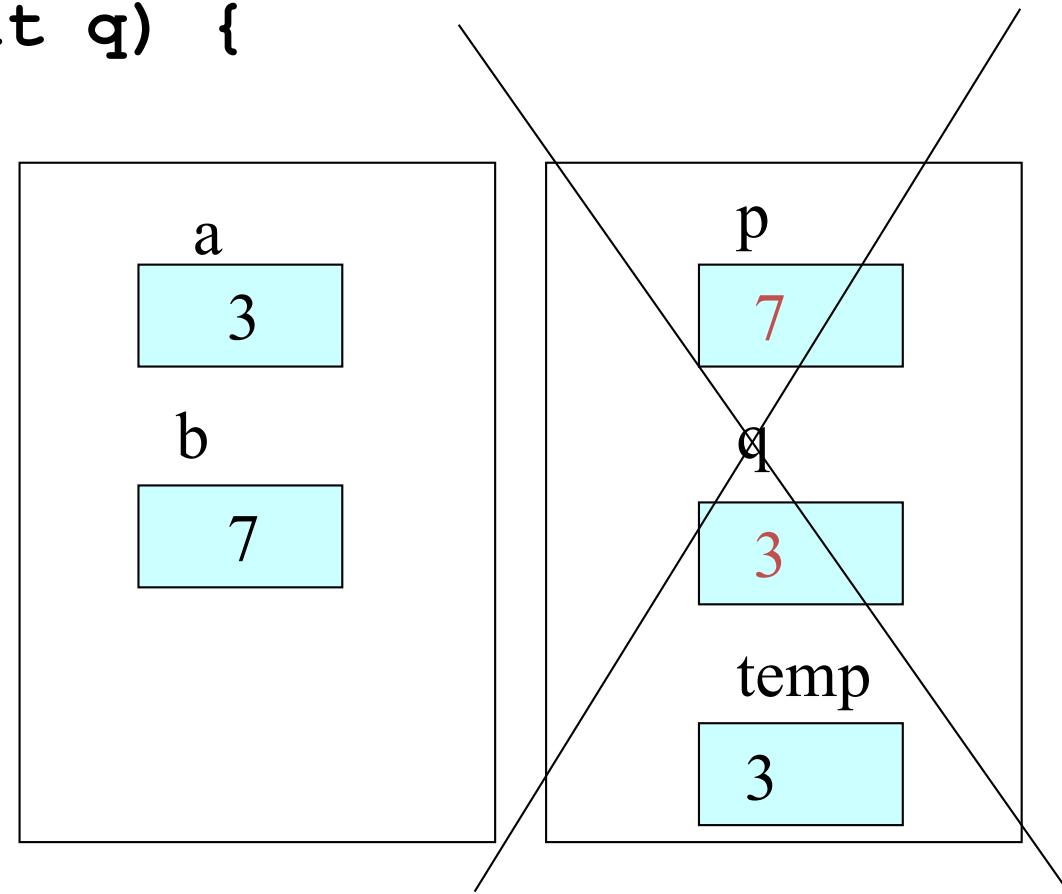


# Example: swap of 2 ints

```
void swap (int p, int q) {  
    int temp;  
    temp = p;  
    p = q;  
    q = temp;  
}
```

In main: swap (a, b)

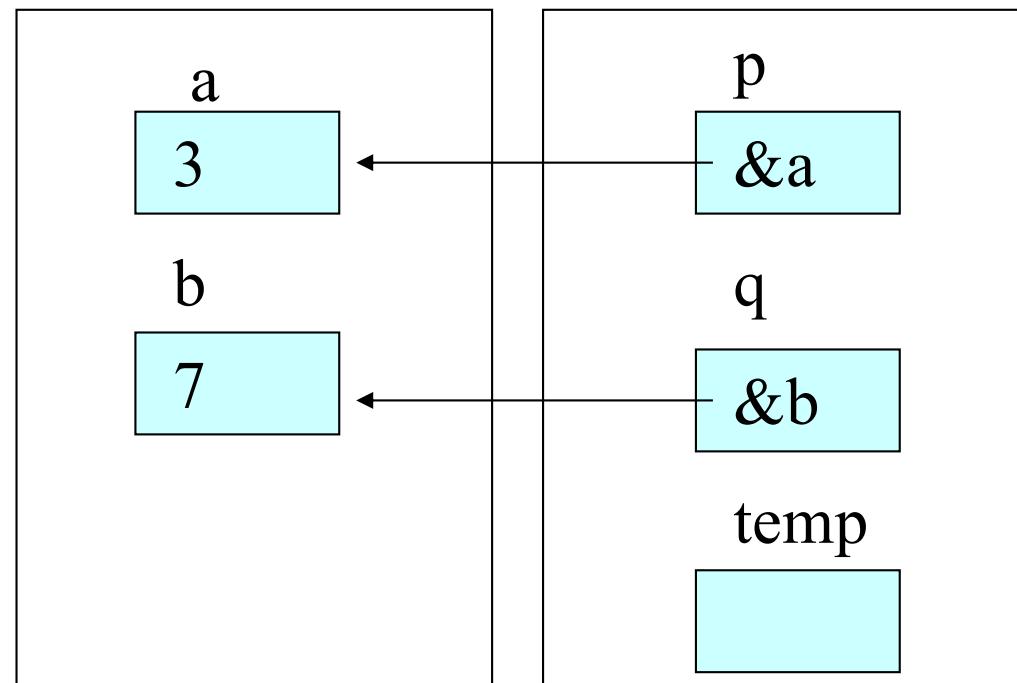
At the end of swap main  
variables are **unchanged!**



# Example: swap of 2 ints

```
void swap (int *p, int *q) {  
    int temp;  
    temp = *p;  
    *p = *q;  
    *q = temp;  
}
```

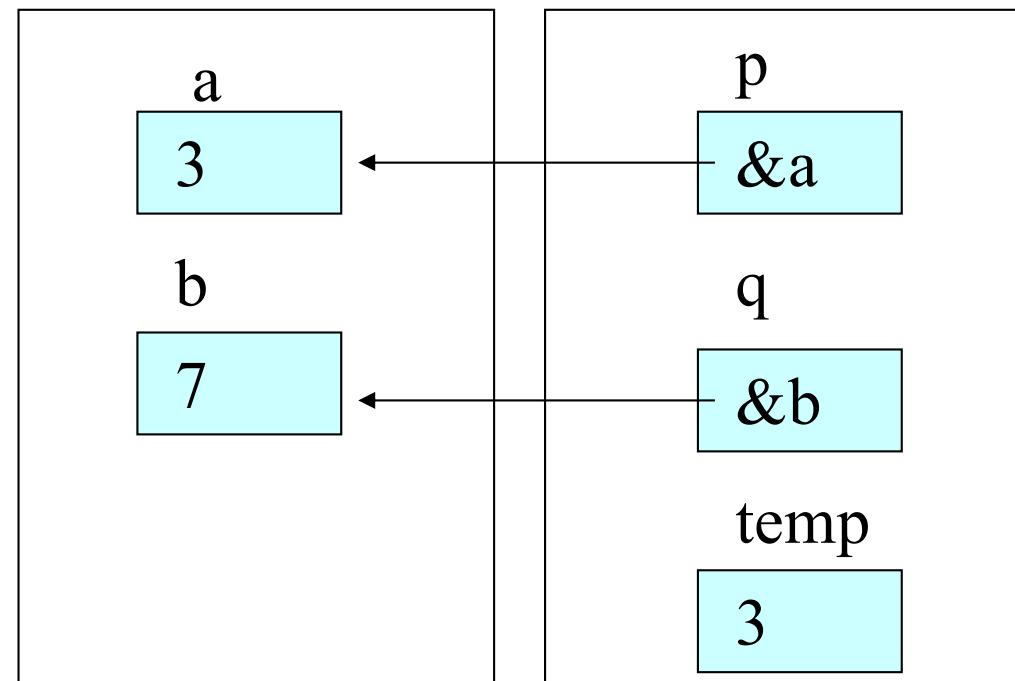
In main: swap (&a, &b)



# Example: swap of 2 ints

```
void swap (int *p, int *q) {  
    int temp;  
    → temp = *p;  
    *p = *q;  
    *q = temp;  
}
```

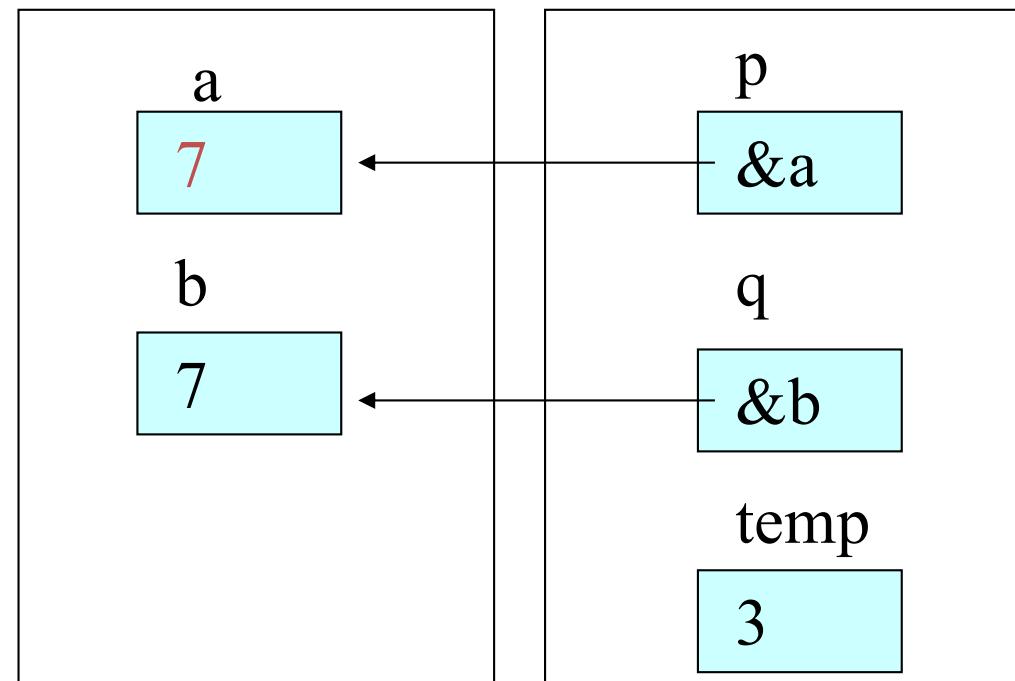
In main: swap (&a, &b)



# Example: swap of 2 ints

```
void swap (int *p, int *q) {  
    int temp;  
    temp = *p;  
    → *p = *q;  
    *q = temp;  
}
```

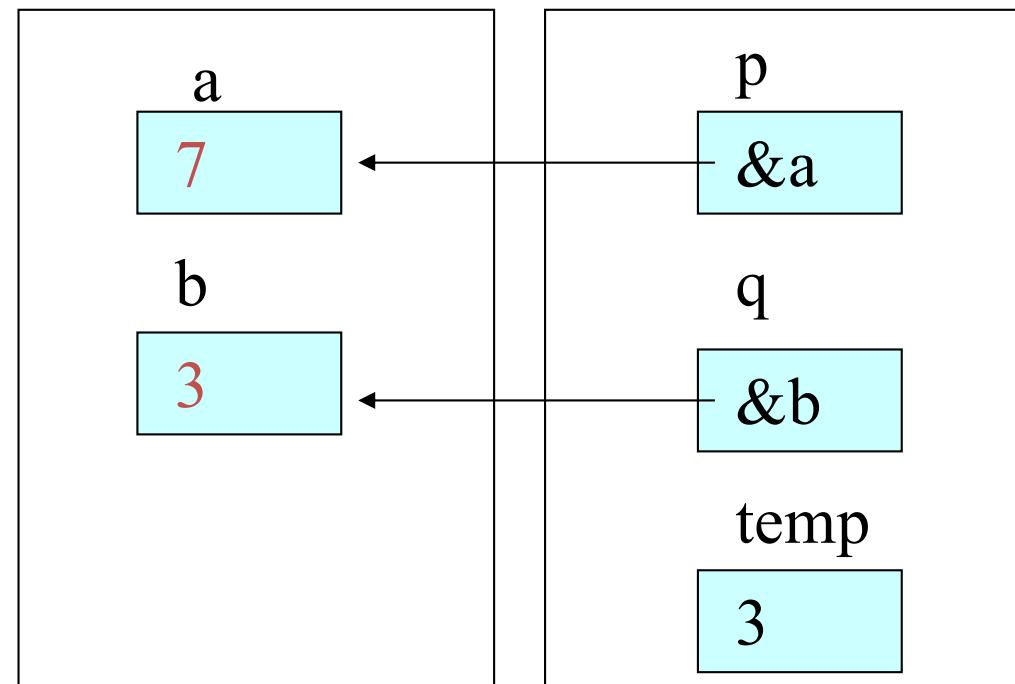
In main: swap (&a, &b)



# Example: swap of 2 ints

```
void swap (int *p, int *q) {  
    int temp;  
    temp = *p;  
    *p = *q;  
    → *q = temp;  
}
```

In main: swap (&a, &b)

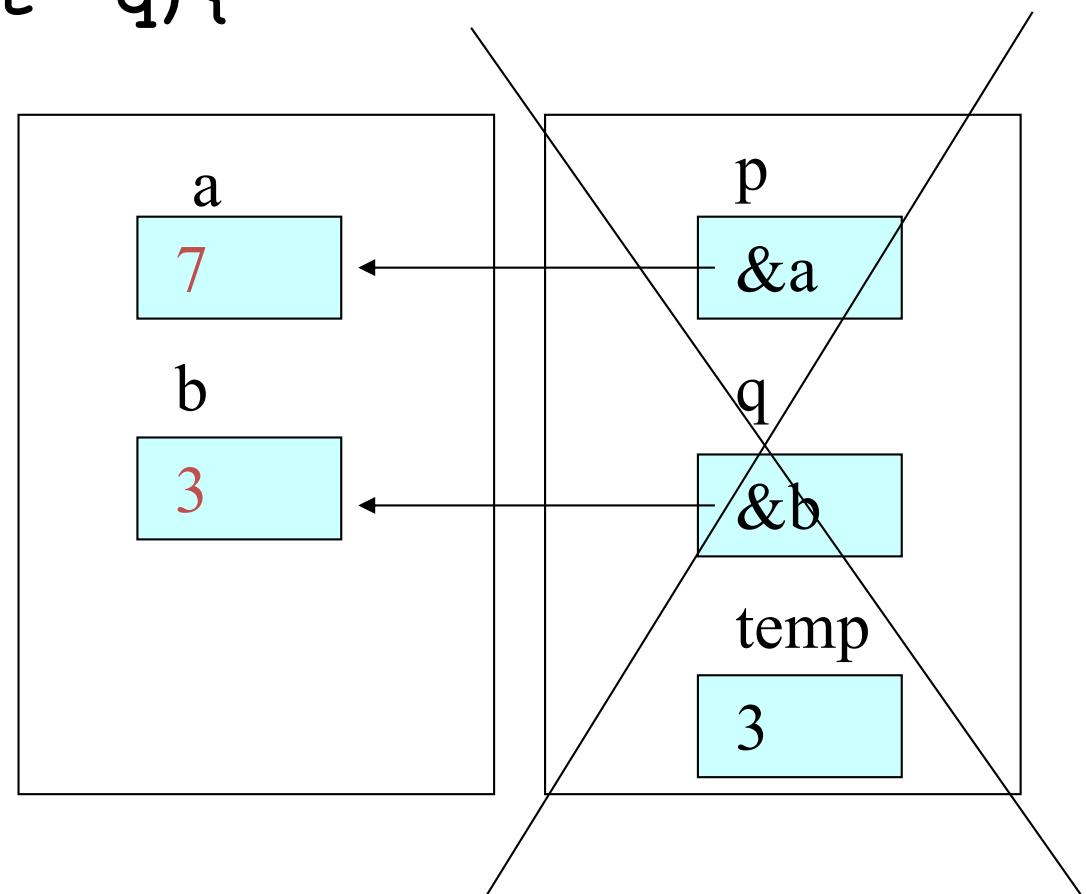


# Example: swap of 2 ints

```
void swap (int *p, int *q) {  
    int temp;  
    temp = *p;  
    *p = *q;  
    *q = temp;  
}
```

In main: swap (&a, &b)

At the end of swap main variables are **changed!**



```
1  
2  
3  
4  
5 #include <iostream>  
6  
7 void cube( int * nPtr ); /* prototype */  
8  
9 void main()  
10 {  
11     int number = 5;  
12  
13     std::cout << "The original value of number is " << number;  
14     cube( &number );  
15     printf( "\nThe new value of number is %d\n" );  
16  
17 }  
18  
19  
20 void cube( int *nPtr )  
21 {  
22     *nPtr = *nPtr * *nPtr * *nPtr;  
23 }
```

**cube** requires a pointer!!

In cube, **\*nPtr** is used (**\*nPtr** is **number**!)

The original value of number is 5  
The new value of number is 125

# From functions to procedures

→ function:

```
int f(int par1)
{
... (compute integer result) ...
return result;
}
```

call:

**y=f(x);**

→ void function:

```
void f(int par1,int *par2)
{
... (compute integer result) ...
*par2=result;
}
```

call:

**f(x, &y);**

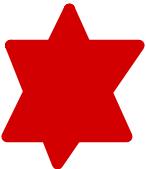
**y gets the proper result value**

# If we need to return multiple values...

```
void f(int in_par1, ..., int in_parn, int *out_par1, int *out_parn,)  
{  
... (compute integer results) ...  
*out_par1=result1;  
...  
*out_parn=resultn;  
}
```

Call (one input parameter, two output parameters):

```
f(x, &y, &z);  
y, z get the proper result values
```



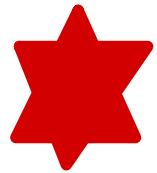
# Pass by value and pass by reference comparison

- Pass by value:

- Requires a lot of time to perform the copy if the parameter is large
- Actual parameter and formal parameter are different
- Cannot return a value to the caller (without a `return` statement!)

- Pass by reference:

- An address is copied  $\Rightarrow$  fixed size  $\Rightarrow$  fast!
- Actual parameter and formal parameter are the same
- Can return a value to the caller



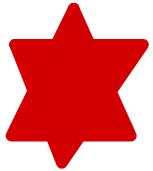
# Summary

Pass	by value	by reference
Property		
Time and Space	Large	Small
Side effects risk	No	<b>Yes</b>
Return value to the caller	No	Yes

# References

---

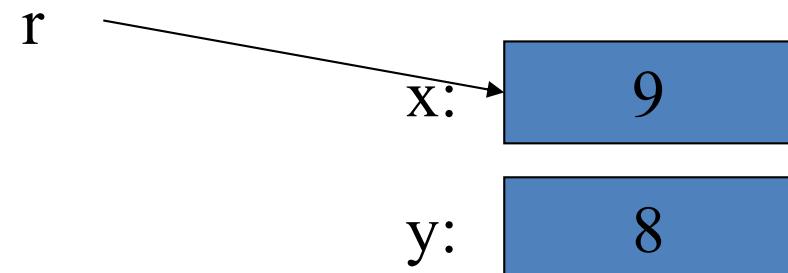
How to make parameter passing a bit easier



# References

- An automatically dereferenced pointer:
  - Or as “an alternative name for an object”
  - A reference is introduced through the & modifier in a variable declaration
  - A reference must be initialized
  - The value of a reference cannot be changed after initialization
    - I.e., you cannot make a reference refer to another object after initialization

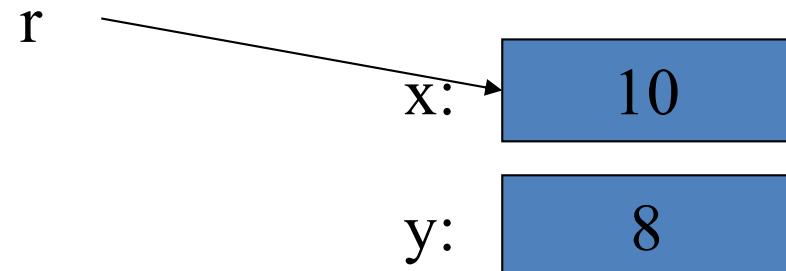
```
int x = 9;  
int y = 8;  
  
int &r = x;
```



# References

- An automatically dereferenced pointer:
  - Or as “an alternative name for an object”
  - A reference is introduced through the & modifier in a variable declaration
  - A reference **must be** initialized
  - The value of a reference **cannot be changed after initialization**
    - I.e., you cannot make a reference refer to another object after initialization

```
int x = 9;  
int y = 8;  
  
int &r = x;  
r = 10; // x = 10!  
r = &y; // error (and so is all other attempts to change what r  
// refers to)
```





# References

- When we initialize a variable, the value of the initializer is copied into the object we are creating
- When we define a reference, instead of copying the initializer value, we bind the reference to its initializer
- Once initialized, a reference remains bound to its initial object
  - There is no way to rebinding a reference to refer to a different object
  - Because of this, references must be initialized
- When we fetch the value of a reference:
  - We are really fetching the value of the object to which the reference is bound
- When we use a reference as an initializer:
  - We are really using the object to which the reference is bound

```
int i = 7;  
int& r = i;  
int &r3 = r; // ok: r3 is bound to the same object as r, i.e., to i
```

// initializes j from the value in the object to which r is bound  
int j = r; // ok: initializes j to the same value as i



# Pointers and references

- A pointer is a compound type that “points to” another type
- Like references, pointers are used for indirect access to other objects
- Unlike a reference, a **pointer is an object in its own right**
  - Pointers can be assigned and copied; a single pointer can point to several different objects over its lifetime
  - Unlike a reference, a **pointer does not need to be initialized at the time it is defined**
- Like other built-in types, pointers have undefined value if they are not initialized. **Be very careful !!!**

# Pointers and references



- & and \*, are used as both an operator in an expression and as part of a declaration
- The **context** in which a symbol is used **determines** what the symbol **means**

```
int i = 42;  
int &r = i;    // & follows a type and is part of a declaration; r is a reference  
int *p;        // * follows a type and is part of a declaration; p is a pointer
```

p = &i; // & is used in an expression as the address-of operator

\*p = i; // \* is used in an expression as the dereference operator

```
int &r2 = *p; // & is part of the declaration; * is the dereference operator  
int *p2 = &i; // * is part of the declaration; & is the address-of operator
```

# C++ pass by reference

- C++ relies on references to implement pass by reference parameters passing mechanism
- This simplifies a lot syntax and notation

DEMO

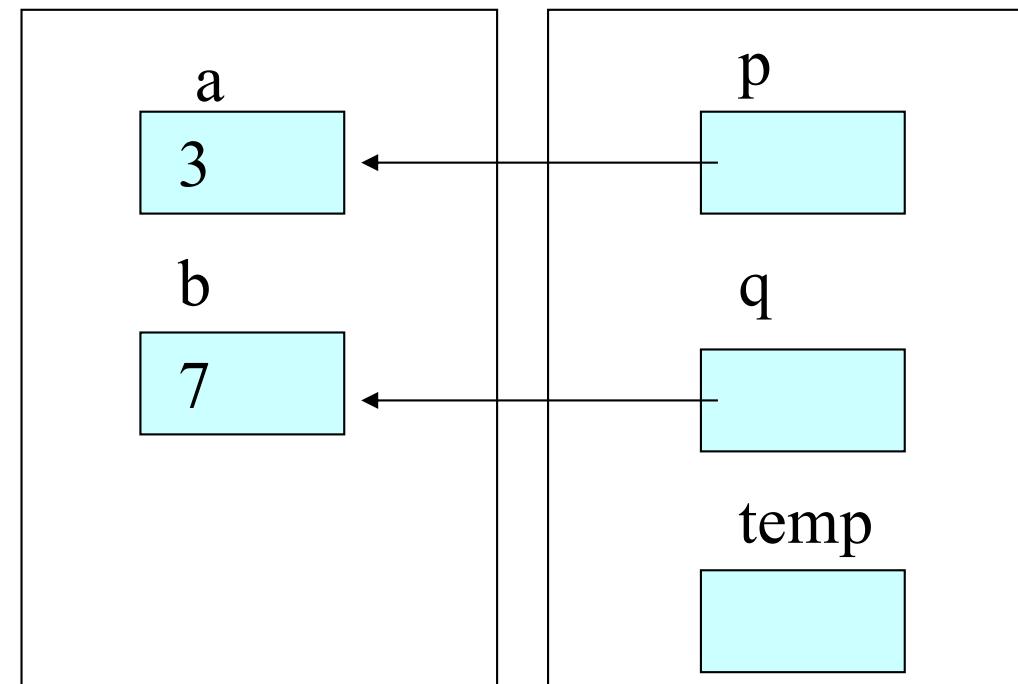


<https://forms.office.com/e/nkG2Y3dLw2>

# Example: swap of 2 ints

```
void swap (int &p, int &q) {  
    int temp;  
    temp = p;  
    p = q;  
    q = temp;  
}
```

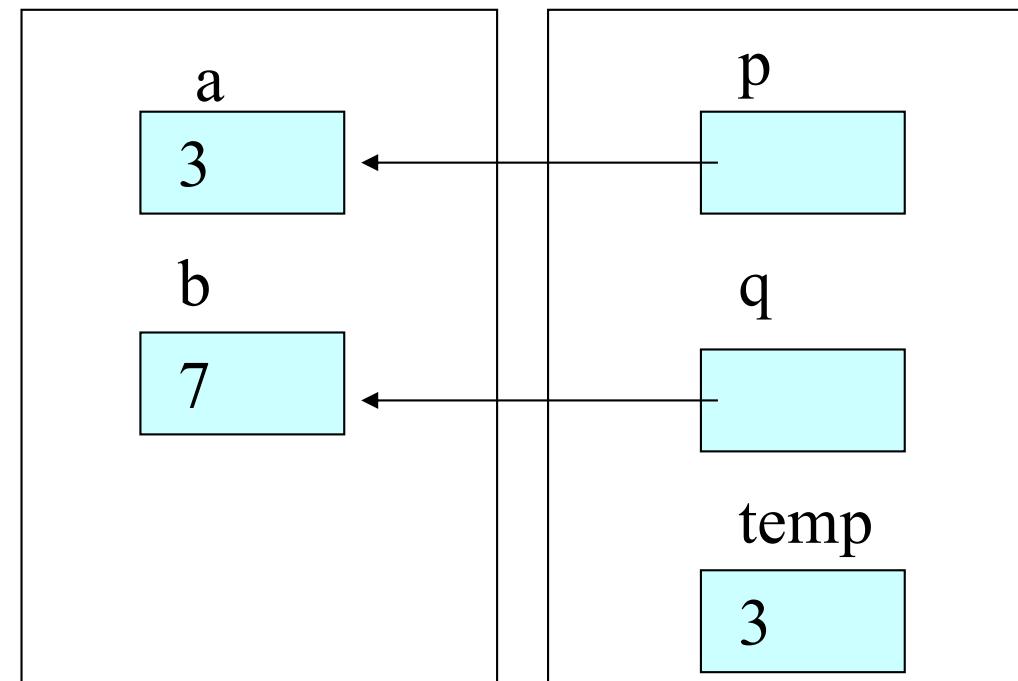
In main: swap(a, b)



# Example: swap of 2 ints

```
void swap (int &p, int &q) {  
    int temp;  
    → temp = p;  
    p = q;  
    q = temp;  
}
```

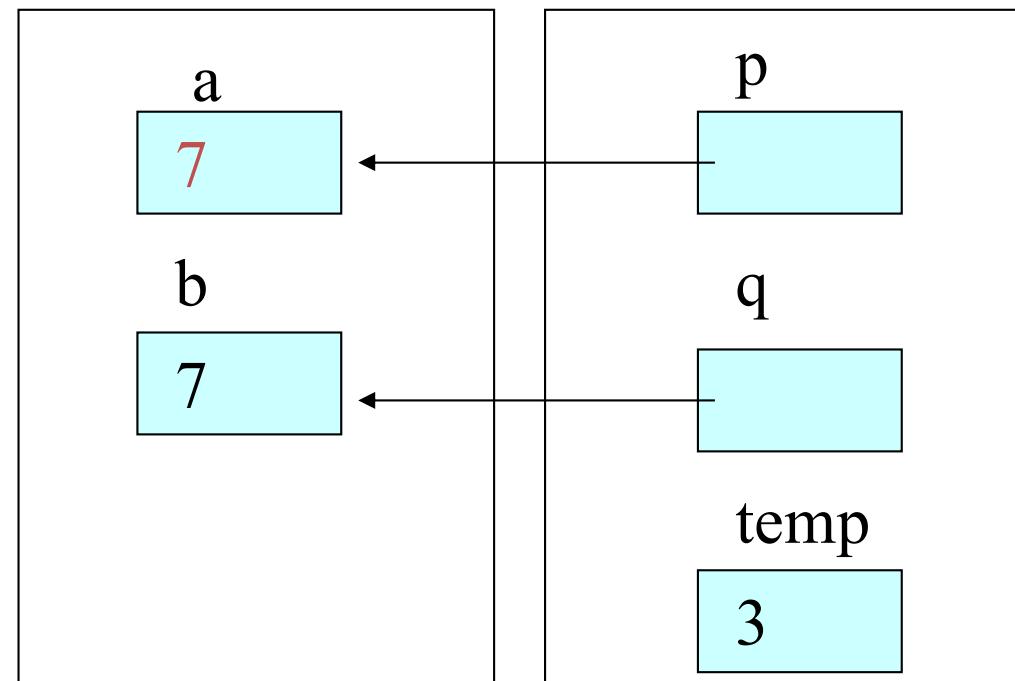
In main: swap(a, b)



# Example: swap of 2 ints

```
void swap (int &p, int &q) {  
    int temp;  
    temp = p;  
    → p = q;  
    q = temp;  
}
```

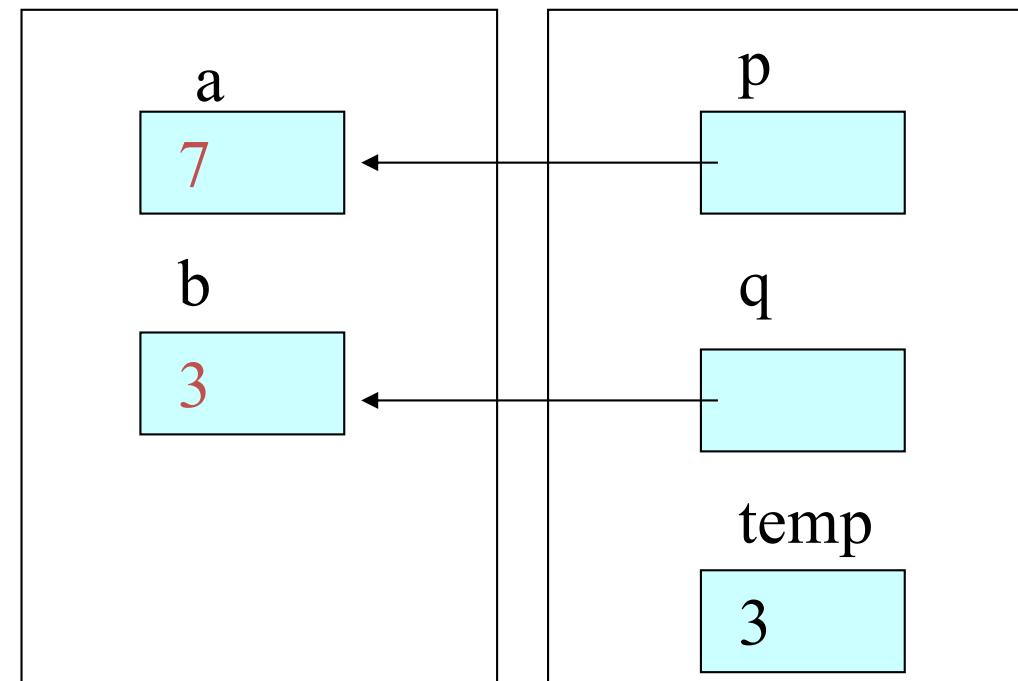
In main: swap(a, b)



# Example: swap of 2 ints

```
void swap (int &p, int &q) {  
    int temp;  
    temp = p;  
    p = q;  
    q = temp;  
}
```

In main: swap(a, b)

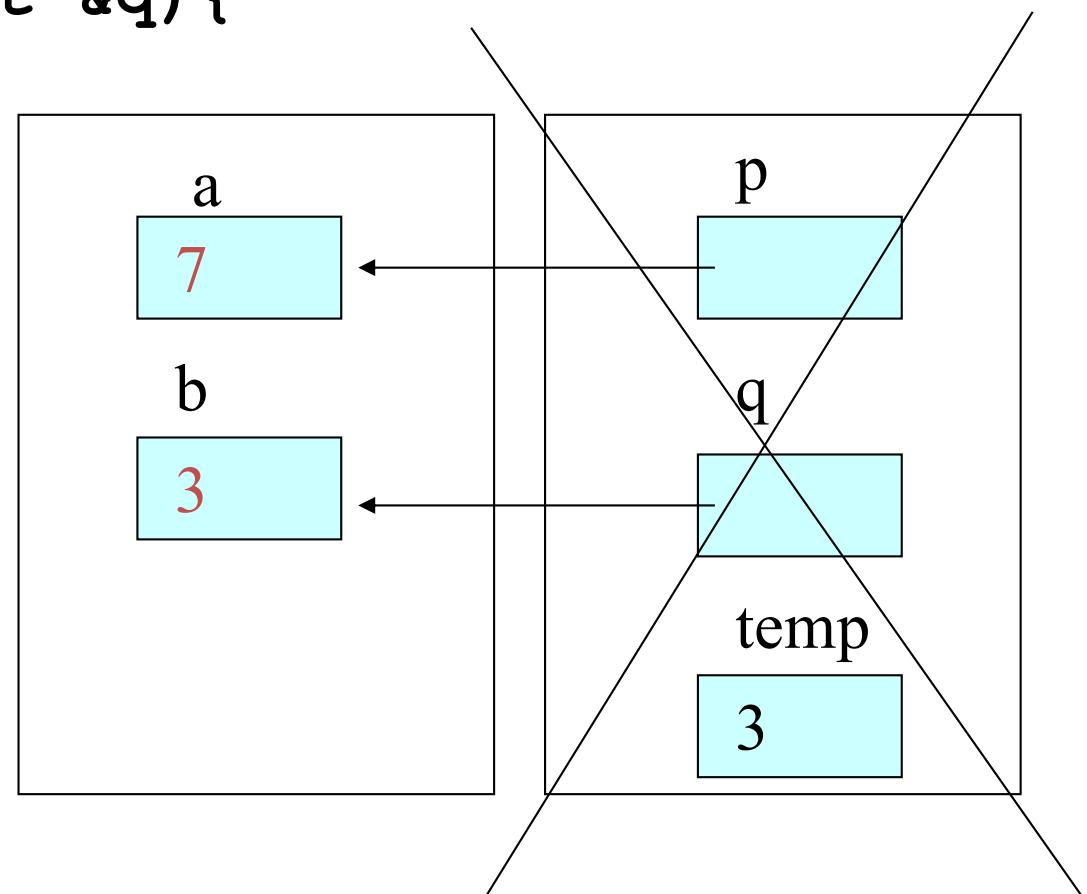


# Example: swap of 2 ints

```
void swap (int &p, int &q) {  
    int temp;  
    temp = p;  
    p = q;  
    q = temp;  
}
```

In main: swap (a, b)

At the end of swap main  
variables are **changed**!



*const* Qualifier

---



# const Qualifier

- We might want to define a variable whose value we know cannot be changed
  - For example, to refer to the size of a buffer size
- We can make a variable unchangeable by defining the variable's type as **const**:

```
const int bufSize = 512;    // input buffer size  
                           // same as constexpr int bufsize=512  
bufSize = 512;             // error: attempt to write to const object
```

- 
- Because we can't change the value of a **const** object after created, it must be initialized

```
const int j = 42; // ok: initialized at compile time  
const int i = get_size(); // ok: initialized at run time  
const int k; // error: k is uninitialized const  
  
j = 47; // error: we try to change a const variable
```

# *const* Qualifier

- By default, *const* objects are local to a file
- When a *const* object is initialized from a compile-time constant, our compiler will usually replace uses of the variable with its corresponding value during compilation
  - The compiler will generate code using the value 512 in the places that our code uses *bufSize*

# References to *const*

- We can bind a reference to an object of a *const* type
- To do so we use a **reference to *const***, which is a reference that refers to a *const* type
- Unlike an ordinary reference, a reference to *const* cannot be used to change the object to which the reference is bound



Const references can be used to pass **large objects in read only** (obtaining the same benefits of C arrays passing + **read only protection**, i.e., no side effects)

ng to

# Example: pass by (const) reference

```
double circ(double *radius) {  
    double res;  
    res = *radius * 3.14 *2;  
    *radius = 7; /* No sense instruction,  
                  let's see what happens to radius */  
    return res;  
}
```

```
// somewhere in the main  
double c;  
double r = 5;  
  
c = circ(&r);  
//Warning! Now r is 7.0
```

```
double circ(const double &radius) {  
    double res;  
    res = radius * 3.14 *2;  
    radius = 7; /* No sense instruction,  
                  let's see what happens to radius */  
    return res;  
}
```

```
// somewhere in the main  
double c;  
double r = 5;  
  
c = circ(r);  
//r is 5.0
```

# Example: pass by (const) reference

```
double circ(double *radius) {  
    double res;  
    res = *radius * 3.14 *2;  
    *radius = 7; /* No sense instruction,  
                  let's see what happens to radius */  
    return res;  
}  
  
// somewhere in the main  
double c;  
double r = 5;  
  
c = circ(&r);  
//Warning! Now r is 7.0
```

```
double circ(const double &radius) {  
    double res;  
    res = radius * 3.14 *2;  
    radius = 7; /* No sense instruction,  
let's see what happens to radius */  
    return res;  
}  
  
// somewhere in the main  
double c;  
double r = 5;  
  
c = circ(r);  
//r is 5.0
```

We get a compiler error here and we need to delete this line of code

# Example: pass by (const) reference

```
double circ(double *radius) {  
    double res;  
    res = *radius * 3.14 *2;  
    *radius = 7; /* No sense instruction,  
        let's see what happens to radius */  
    return res;  
}
```

```
// somewhere in the main  
double c;  
double r = 5;  
  
c = circ(&r);  
//Warning! Now r is 7.0
```

What about `circ(22.0);?`

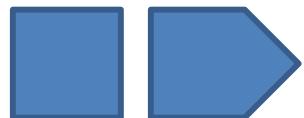
```
double circ(const double &radius) {  
    double res;  
    res = radius * 3.14 *2;  
    return res;  
}
```

```
// somewhere in the main  
double c;  
double r = 5;  
  
c = circ(r);  
//r is 5.0
```

A temporary const object is created, initialized with 22.0 and the function can be executed

# Call by Value vs. by Call Reference

## Play video



*pass by reference*

cup =

fillCup( )

*pass by value*

cup =

fillCup( )

# Guidance for passing variables



- Use call-by-value for very small objects (base types!)
- Use call-by-const-reference for large objects
- Use call-by-reference only when you must return a result rather than modify an object through a reference argument

- For example:

```
class Image { /* objects are potentially huge */ };
void f(Image i); ... f(my_image); // oops: this could be s-l-o-o-o-w
void f(Image& i); ... f(my_image); // no copy, but f() can modify
                                    // my_image
void f(const Image& i); ... f(my_image); // f() won't mess with
                                         // my_image
```

# Variables scope

---



# Scope

- The scope of an identifier is the portion of the program in which the identifier can be referenced
  - Some identifiers can be referenced throughout the program
  - Others can be referenced from only portions of a program
- Example:

When we declare a local variable in a block (e.g., in a for loop), it can be referenced only in that block or in blocks nested within that block



# Methods of variable creation



- Global variable
  - declared very early stage
  - available always and from anywhere
  - created at the start of the program, and lasts until the end, stored in the **static data**
  - difficult to debug

Never use global variables!

**-5 points** at exams!

memory layout:





# Methods of variable creation

- Local on the fly variables

- simply created when they are needed
- only available from within the routine/block in which they were created, stored in the stack
- easy to debug

```
for (unsigned j = 0; j < 10; ++j) {  
    // do something  
}
```

memory layout:





# Methods of variable creation

- **Local defined variable**

- created before they are needed
- only available in the routine in which they were created
- easy to debug, stored in the **stack**
- the most usually favored method

```
void f (int x) {  
    // do something  
}
```

```
int main () {  
    int y = 7;  
    // etc.  
}
```

memory layout:



# A program written in C++

```
// This program calculates the number calories in a cheese sandwich

#include <iostream>

const int BREAD = 63;           // global constant
const int CHEESE = 106;         // global constant
const int MAYONNAISE = 49;       // global constant
const int PICKLES = 25;          // global constant

int main()
{
    int totalCalories;          // local variable
    totalCalories = 2 * BREAD + CHEESE + MAYONNAISE + PICKLES;
    cout << "There were " << totalCalories;
    cout << " calories in my lunch yesterday." << endl;
    return 0;
}
```

Global constant  
are ok!

# Function definition and function prototype

```
#include <iostream>

int square (int);      // Function prototype

int main()
{
    for (int x = 0; x < 10; x++)
        cout << square(x) << ' ';
    cout << endl;
    return 0;
}

int square (int y)      // Function definition
{
    return y * y;          Local variable
}
```

# Global and Local declarations

```
#include <iostream>
void func( float );
const int a = 17;           // global constant
int b;                     // global variable
int c;                     // global variable
int main()
{
    b = 4;                 // assignment to global b
    c = 6;                 // assignment to global c
    func(42.8);
    return 0;
}
void func( float c) // prevents access to global c
{
    float b;               // prevent access to global b
    b = 2.3;               // assignment to local b
    cout << " a = " << a;
    cout << " b = " << b;
    cout << " c = " << c;
}
```

Be careful with  
variables scope (and  
name hiding)

Values:  
a = 17 b = 4 c = 6

Output:  
a = 17 b = 2.3 c = 42.8

# Explanation

- In this example, *func* accesses global constant *a*
- However, *func* declares its own local variable *b* and parameter *c*
- Local variable *b* takes precedence over global variable *b*, effectively hiding global variable *b* from the statements in function *func*
- Function parameter acts like local variable



# Lifetime of a variable

- Local defined variables: variables declared inside a function or variables declared as function parameter
- When you will call the function, memory will be allocated for all local variables defined inside the function
- Finally, memory will be deallocated when the function exits
- The period of time a variable will be “alive” while the function is executing is called “lifetime” of this variable

# Another example - Scopes nest

```
int x; // global variable – avoid those where you can
int y; // another global variable

void f();

int main(){
    x = 8; y = 3;
    f();
    cout << x << ' ' << y << '\n'; // what will cout print?
}

void f()
{
    int x; // local variable (Note – now there are two x's)
    x = 6; // local x, not the global x
    {
        int x = y; // another local x, initialized by the global y
        // (Now there are three x's)
        ++x; // increment the local x in this scope
    }
    // what is the value of x here?
    y++;
}
```

**// avoid such complicated nesting and hiding: keep it simple!**

DEMO



<https://forms.office.com/e/Q0H5sddmKu>