

7



```
#include <iostream>
using namespace std;
void myfun( int a ) {
    a++;
    cout << "inside myfun, a=" << a << endl;
} // end of myfun()
int main() {
    int a = 7;
    cout << "before calling myfun, a=" << a << endl;
    myfun( a );
    cout << "after calling myfun, a=" << a << endl;
} // end of main()</pre>
```

```
cis15-spring2010-parsons-lectIII.3
```

```
Call by value
```

- With *call by value*, the *value* of each argument is copied to a local variable within the function
- When the function ends, the program control returns to wherever the function was called from, and the memory allocated within the function returns to the program's memory stack
- Even if the values of the local arguments within the function changed during the execution of the function, the values that were used to invoke the function do not change

# • The output is:

before calling myfun, a=7
inside myfun, a=8
after calling myfun, a=7

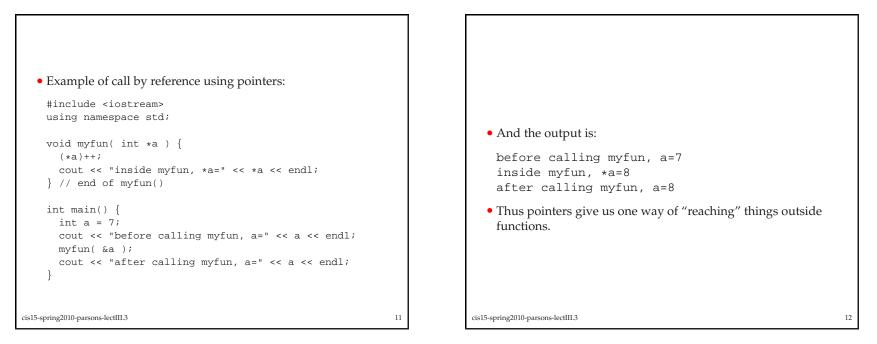
cis15-spring2010-parsons-lectIII.3

# Call by reference

- With *call by reference*, the *address* of each argument is copied to a local variable within the function
- When the function ends, the program control returns to wherever the function was called from, and the memory allocated within the function returns to the program's memory stack
- Because the local arguments are <u>addresses</u>, any changes that were made to the values stored at these address locations during the execution of the function *are retained* when the function ends

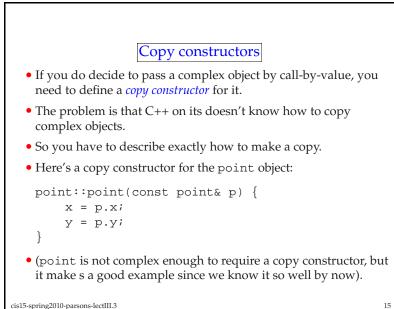
10

- in C++, there are two ways to implement call by reference:
  - using pointers; and
  - using references.





```
#include <iostream>
    using namespace std;
    void myfun( int &a ) {
      a++;
      cout << "inside myfun, a=" << a << endl;</pre>
    } // end of myfun()
    int main() {
      int a = 7;
      cout << "before calling myfun, a=" << a << endl;</pre>
      myfun( a );
      cout << "after calling myfun, a=" << a << endl;</pre>
cis15-spring2010-parsons-lectIII.3
```



# Why use call-by-reference?

- We use call-by-reference for *efficiency*.
- Call-by-value requires the computer to copy the parameters before passing them to the function.
- This is fine if the parameters are a few chars or doubles.
- But in C++ we might call a function on a complex object that holds many many bytes of data.
- It is far more efficient, in both memory and time, to pass a pointer or a reference to such an object than to copy it.
- However, you have to be very careful when you do this otherwise you may get odd things happening to your program.

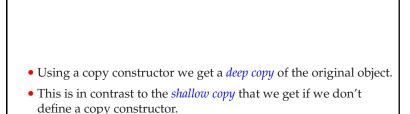
14

16

cis15-spring2010-parsons-lectIII.3

13

- C++ knows this is a copy constructor by the signature.
- There is no return type (just like a constructor).
- The only argument is a reference to an object of the same class as the constructor is defined for.
- The p that is the argument of the copy constructor is the object being copied.
- What the copy constructor has to do is to say how to set the value of every attribute of the object.
- In the example from point, we are saying that to make a copy of p copy the attribute p. x into the attribute x of the copy, and similarly for y.



- Roughly speaking, if an object includes a pointer, we need to make a deep copy of the object.
- For a more complex example of a copy constructor, see the example program dynamic-stack.cpp.

```
cis15-spring2010-parsons-lectIII.3
```

• Thus within the context of a function header definition, the following two statements are equivalent:

int sum( int A[], ... ) { ... }

### and

```
int sum( int *A, ... ) { ... }
```

# but not in other contexts!

• This explains the function headers you see in some of the C++ libraries.

# Passing arrays to functions • Given the following example: int sum( int A[], int n ) { int s=0; for ( int i=0; i<n; i++ ) s += A[i]; return( s ); } // end of sum() • When the array A is passed to the function sum(), it is passed using call-by-value on its base address (i.e., the address of A[0] • However, passing an address call-by-value is the same as</pre>

passing the thing that is addressed call-by-reference.

18

20

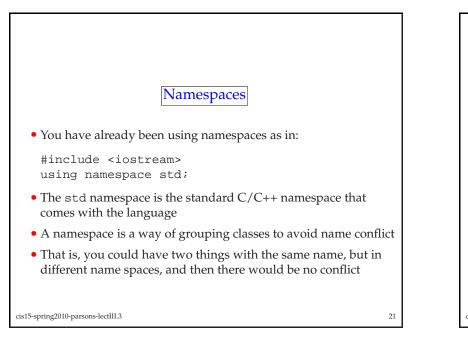
cis15-spring2010-parsons-lectIII.3

17

19

• And the output is:

before calling myfun, a=7
inside myfun, a=8
after calling myfun, a=8



• Declaration of classes within a namespace looks like this: namespace myspace {

```
class myclass1 { ... };
```

```
class myclass2 { ... };
```

- $\} \setminus \$  end of namespace
- Note that when you define a namespace in a header file, you do not need to use the . h in the include statement:
- #include <iostream>
  using namespace std;
  versus

#include <time.h>

• The first include statement is part of a namespace; the second is not

22

cis15-spring2010-parsons-lectIII.3

23

# Summary

- This lecture was mainly concerned with call by reference.
  - We recalled call-by-value.
  - We looked at call-by-reference using pointers.
  - We intrduced references; and
  - We looked at call-by-reference using references
- We also looked briefly at namespaces and passing arrays to functions.