CS 106B
Lecture 14: Pointers

Wednesday, May 3, 2017

Programming Abstractions
Spring 2017
Stanford University
Computer Science Department

Lecturer: Chris Gregg

reading:
Programming Abstractions in C++, Chapter 11

Today's Topics

- Logistics
  - Midterm tomorrow night.
  - You will be able to take the exam on computer or on paper. Make sure your battery is charged!

- Introduction to Pointers
  - What are pointers?
  - Pointer Syntax
  - Pointer Tips
  - Pointer Practice
  - Binky
- Back to classes
  - The copy constructor ~~and the assignment overload~~

C++ Challenge

- Challenge #1: Write a swap function:

```
? swap( ? ) {
    ...
}

int main() {
    // swap the two variables below, using a swap function
    int a = 5;
    int b = 12;
    swap ( ? );
    // at this point, a should equal 12 and b should equal 5
}
```

C++ Challenge

- Can we write a swap function in C++ using what we know already? Yes, we can!

```
void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

int main() {
    // swap the two variables below, using a swap function
    int a = 5;
    int b = 12;
    swap (a, b);
    // at this point, a equals 12 and b equals 5
}
```

C++ Challenge

- So it turns out that references are a nice C++ feature, but they abstract away some of the lower-level details that we might want to know about.

- In order for our swap function to work, we must have access to the original elements.

- This starts to fall under the category of "memory management"

- As a close relative to C, C++ gives us access to all of C's low-level functionality.

Introduction to Pointers

- The next major topic is about the idea of a *pointer* in C++. We need to use pointers when we create data structures like Vectors and *Linked Lists* (which we will do next week!)
- Pointers are used heavily in the C language, and also in C++, though we haven't needed them yet.

- Pointers delve under the hood of C++ to the memory system, and so we must start to become familiar with how memory works in a computer.

Introduction to Pointers

- The memory in a computer can be thought of simply as a long row of boxes, with each box having a value in it, and an index associated with it.
- If this sounds like an array, it's because it is!
- Computer memory (particularly, Random Access Memory, or RAM) is just a giant array. The "boxes" can hold different types, but the numbers associated with each box is just a number, one after the other:

values (ints):	7	2	8	3	14	99	-6	3	45	11
associated index:	0	1	2	3	4	5	6	7	8	9

values (strings):	cat	dog	apple	tree	shoe	hand	chair	light	cup	toe
associated index:	10	11	12	13	14	15	16	17	18	19

Introduction to Pointers

- In C++, we just call those boxes variables, and we call the associated indices *addresses*, because they can tell us where the variable is located (like a house address).

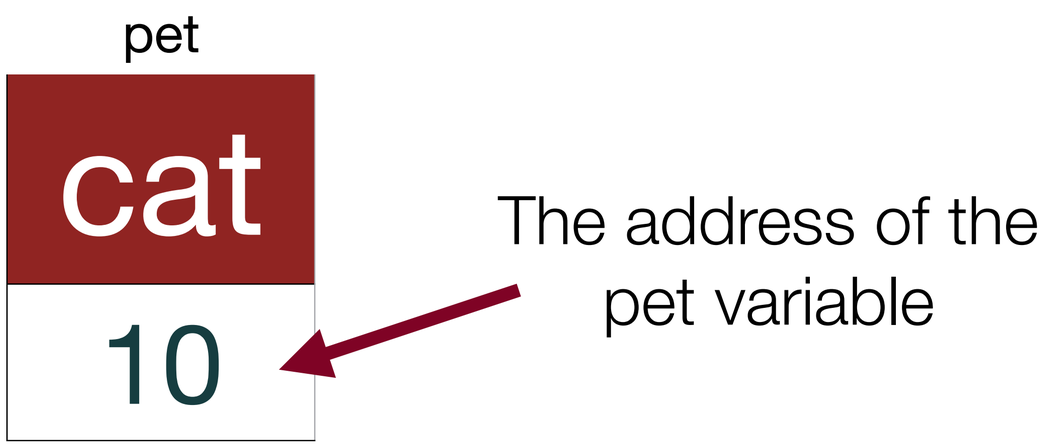
variable:	cat	dog	apple	tree	shoe	hand	chair	light	cup	toe
address:	10	11	12	13	14	15	16	17	18	19

```
string pet = "cat";
```

- What is the address of the **`pet`** variable?

10

- The operating system determines the address, not you! In this case it is 10, but it could be any other address in memory.



Introduction to Pointers

- Guess what? If we store that **memory address** in a different variable, it is called a ***pointer***.

```
string pet = "cat";
```

Some other variable:

So, what is a pointer?

A memory address!

Introduction to Pointers

What is a pointer??

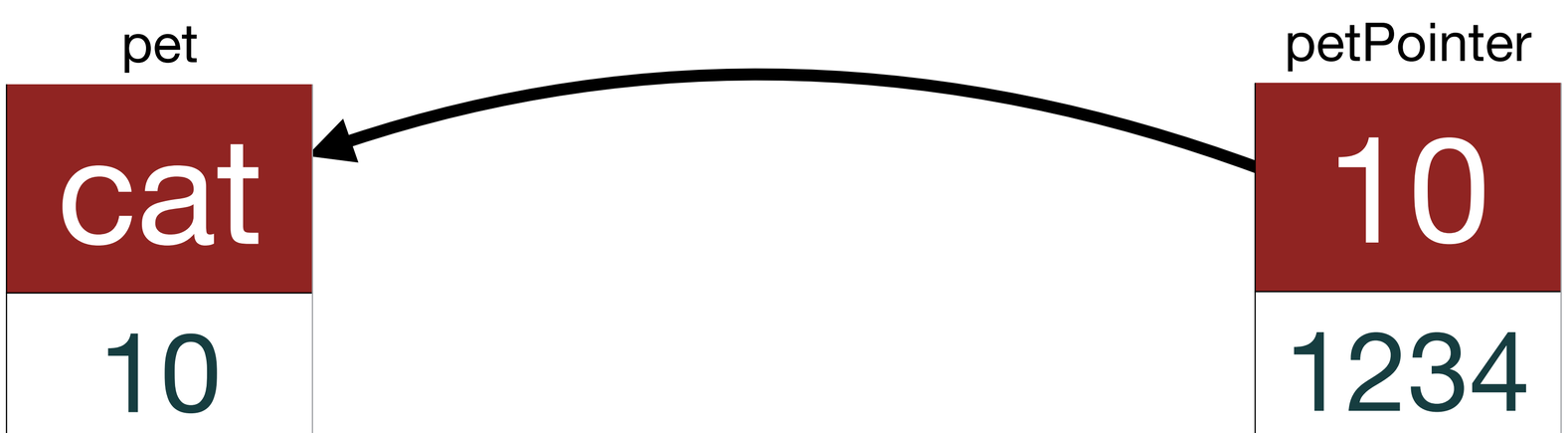
a memory address!

Introduction to Pointers

- We really don't care about the actual memory address numbers themselves, and most often we will simply use a visual "pointer" to show that a variable points to another variable:

```
string pet = "cat";
```

petPointer variable

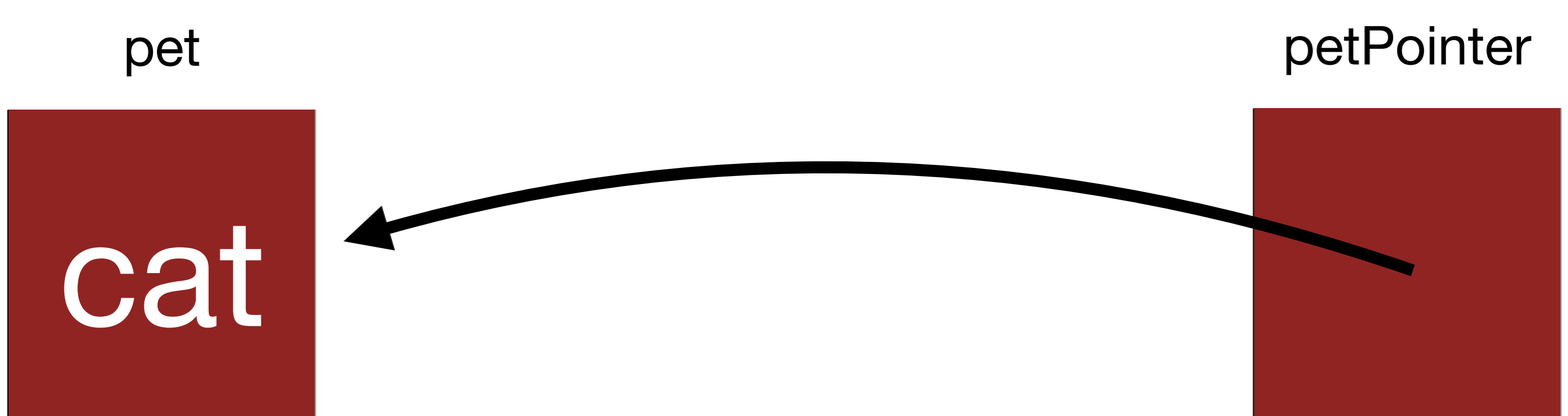


Introduction to Pointers

- We really don't care about the actual memory address numbers themselves, and most often we will simply use a visual "pointer" to show that a variable points to another variable:

```
string pet = "cat";
```

pet_pointer variable



Introduction to Pointers

What you need to know about pointers:

- Every location in memory, and therefore every variable, has an address.
- Every address corresponds to a unique location in memory.
- The computer knows the address of every variable in your program.
- Given a memory address, the computer can find out what value is stored at that location.
- While addresses are just numbers, C++ treats them as a separate type. This allows the compiler to catch cases where you accidentally assign a pointer to a numeric variable and vice versa (which is almost always an error).

Pointer Syntax

Pointer syntax can get tricky. We will not go too deep -- you'll get that when you take cs107!

Pointer Syntax #1: To declare a pointer of a particular type, use the "`*`" (asterisk) symbol: 

```
string *petPtr;  // declare a pointer (which will hold a
                 //    memory address) to a string
int *agePtr;     // declare a pointer to an int
char *letterPtr; // declare a pointer to a char
```

The type for petPtr is a "`string *`" and *not* a `string`. This is important! A pointer type is distinct from the pointee type.

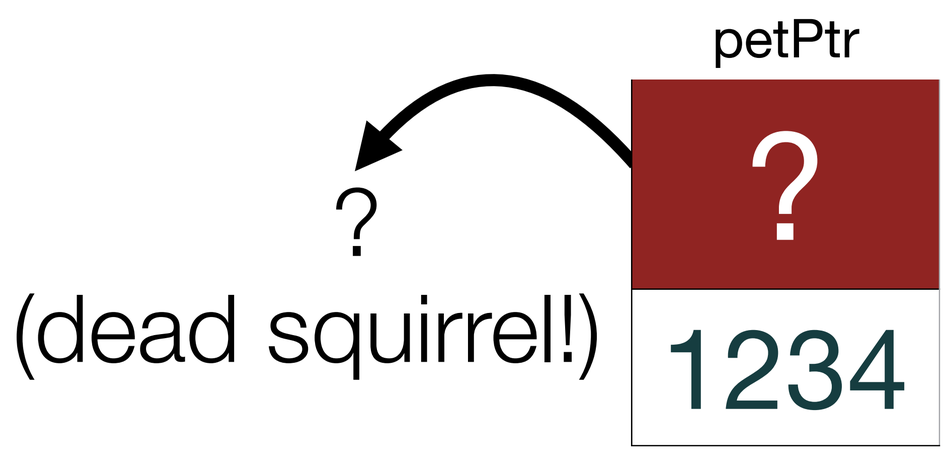
Pointer Syntax

Pointer Syntax #2: To *get the address of another variable*, use the "**&**" (ampersand) character:

Pointer Syntax

Pointer Syntax #2: To *get the address of another variable*, use the "**&**" (ampersand) character:

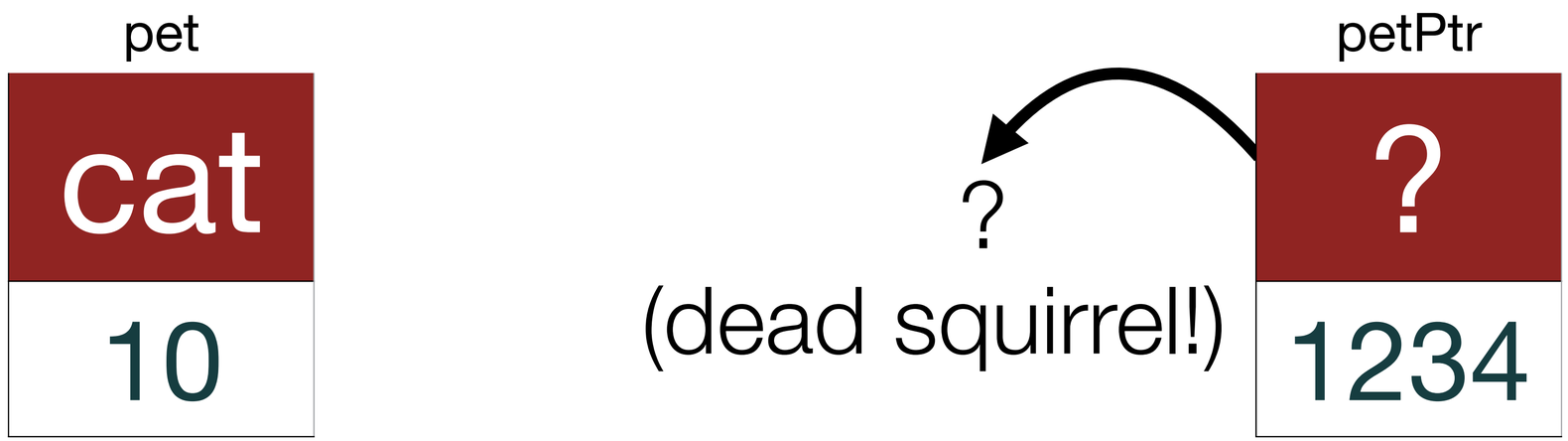
```
string *petPtr; // declare a pointer (which will hold a
                //    memory address) to a string
```



Pointer Syntax

Pointer Syntax #2: To *get the address of another variable*, use the "**&**" (ampersand) character:

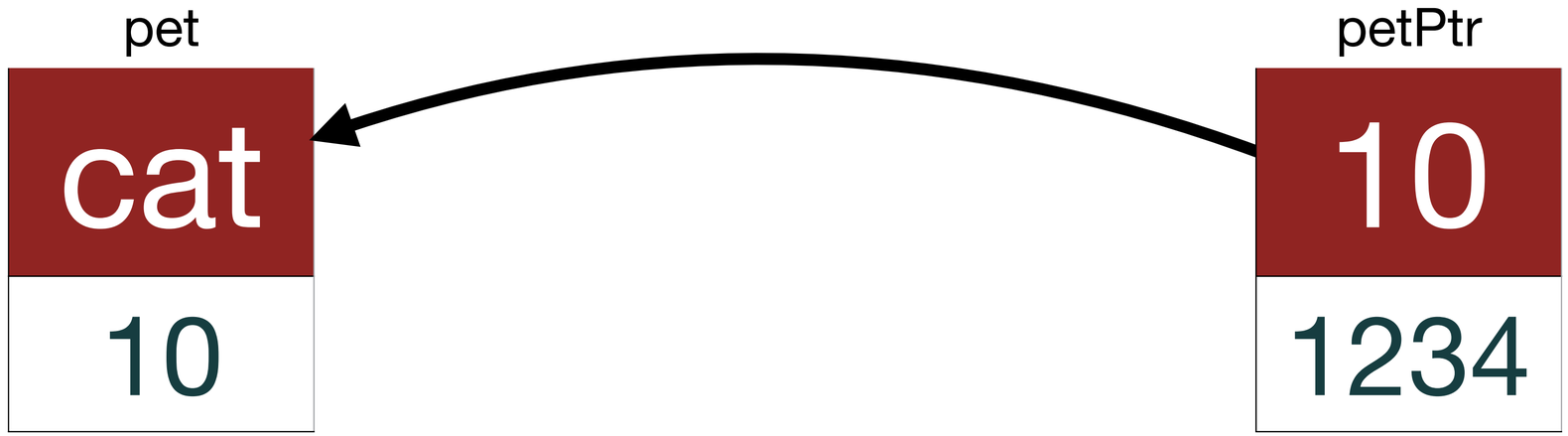
```
string *petPtr; // declare a pointer (which will hold a
                //    memory address) to a string
string pet = "cat"; // a string variable
```



Pointer Syntax

Pointer Syntax #2: To *get the address of another variable*, use the "**&**" (ampersand) character:

```
string *petPtr; // declare a pointer (which will hold a
                //    memory address) to a string
string pet = "cat"; // a string variable

petPtr = &pet; // petPtr now holds the address of pet
```



Pointer Syntax

Pointer Syntax #2: To *get the address of another variable*, use the "**&**" (ampersand) character:

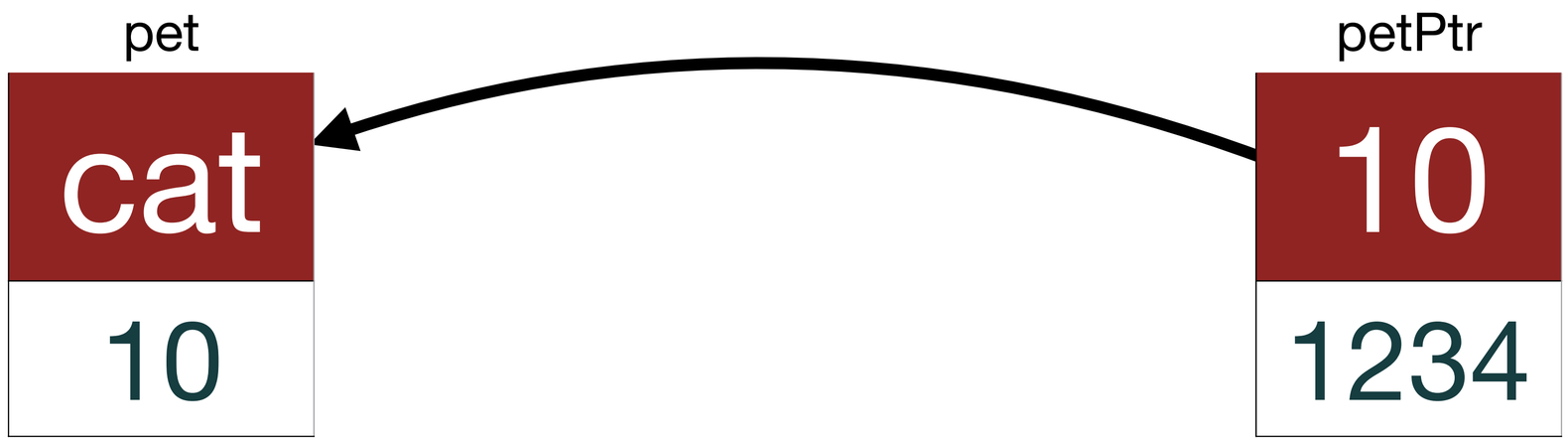
```
string *petPtr; // declare a pointer (which will hold a
                //    memory address) to a string
string pet = "cat"; // a string variable

petPtr = &pet; // petPtr now holds the address of pet
```

you almost **never** need to do this in 106B!!!

Pointer Syntax

Pointer Syntax #3: To *get value of the variable a pointer points to*, use the "*****" (asterisk) character (in a different way than before!):

```
string *petPtr; // declare a pointer to a string
string pet = "cat"; // a string variable
petPtr = &pet; // petPtr now holds the address of pet
cout << *petPtr << endl; // prints out "cat"
```



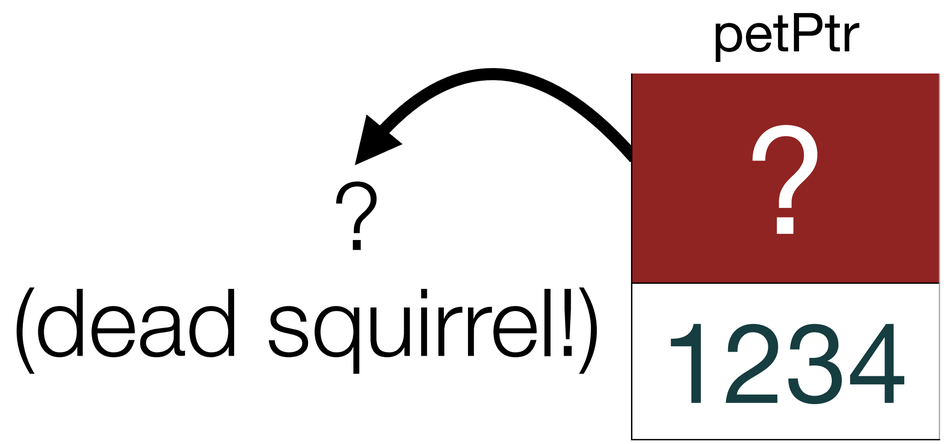
This is called "dereferencing" the pointer: the asterisk says, "go to where the pointer is pointing, and return the *value* stored there"

Pointer Tips

Pointer Tip #1: To ensure that we can tell if a pointer has a valid address or not, set your declared pointer to **`NULL`**, which means "no valid address" (it actually is just 0 in C++).

Instead of this:

```
string *petPtr; // declare a pointer to
                // a string with a dead squirrel
```

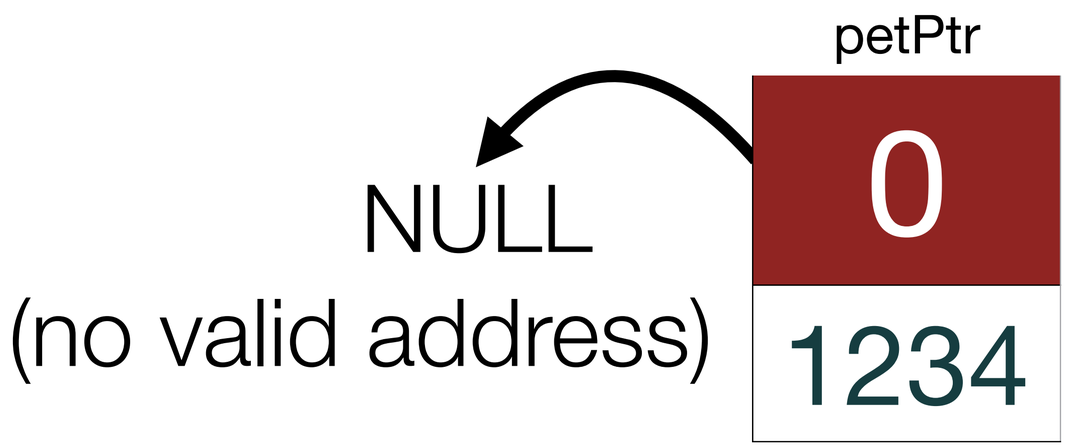


Pointer Tips

Pointer Tip #1: To ensure that we can tell if a pointer has a valid address or not, set your declared pointer to **`NULL`**, which means "no valid address" (it actually is just 0 in C++).

Do this:

```
string *petPtr = NULL; // declare a pointer to
                       // a string that points to NULL
```



Pointer Tips

Pointer Tip #2: If you are unsure if your pointer holds a valid address, you should check for **`NULL`**

Do this:

```
void printPetName(string *petPtr) {
    if (petPtr != NULL) {
        cout << *petPtr << endl; // prints out the value
                                 // pointed to by petPtr
                                 // if it is not NULL
    }
}
```

Pointer Practice

These little boxes we draw to show the memory are so, so important to understanding what is happening. Always draw boxes when learning pointers!

```
int *nPtr = NULL;
```

What type does this pointer point to?
What should we draw?

Pointer Practice

These little boxes we draw to show the memory are so, so important to understanding what is happening. Always draw boxes when learning pointers!

```
int *nPtr = NULL;
```

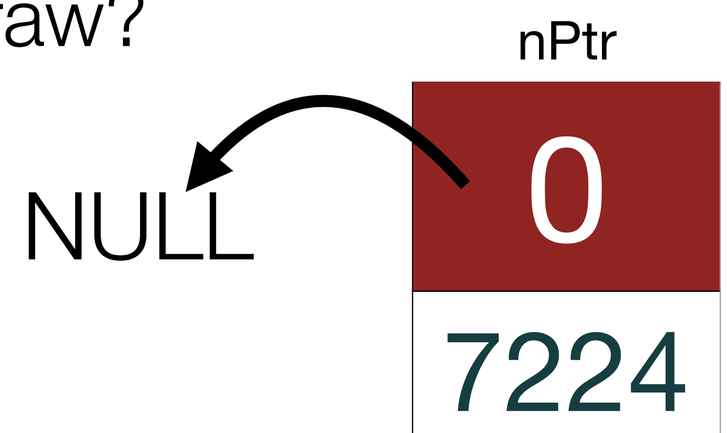
What type does this pointer point to? **an int**
What should we draw?

Pointer Practice

These little boxes we draw to show the memory are so, so important to understanding what is happening. Always draw boxes when learning pointers!

```
int *nPtr = NULL;
```

What type does this pointer point to? **an int**
What should we draw?

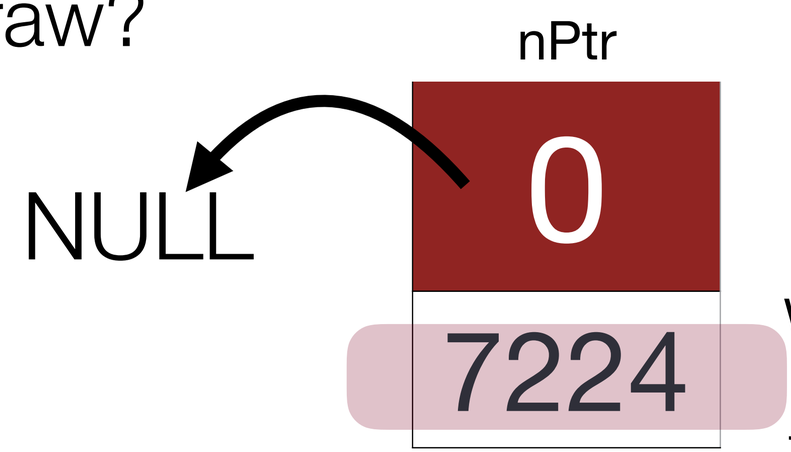


Pointer Practice

These little boxes we draw to show the memory are so, so important to understanding what is happening. Always draw boxes when learning pointers!

```
int *nPtr = NULL;
```

What type does this pointer point to? **an int**
What should we draw?



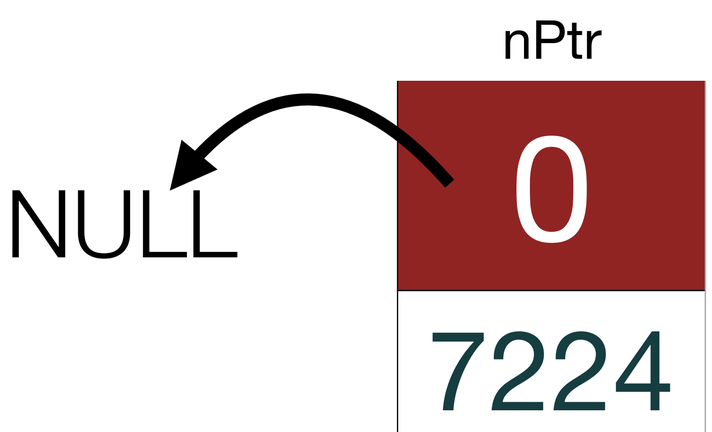
We don't care what this number is, just that it tells us where **`nPtr`** is in memory.

Pointer Practice

These little boxes we draw to show the memory are so, so important to understanding what is happening. Always draw boxes when learning pointers!

```
int *nPtr = NULL;
int n = 16;
```

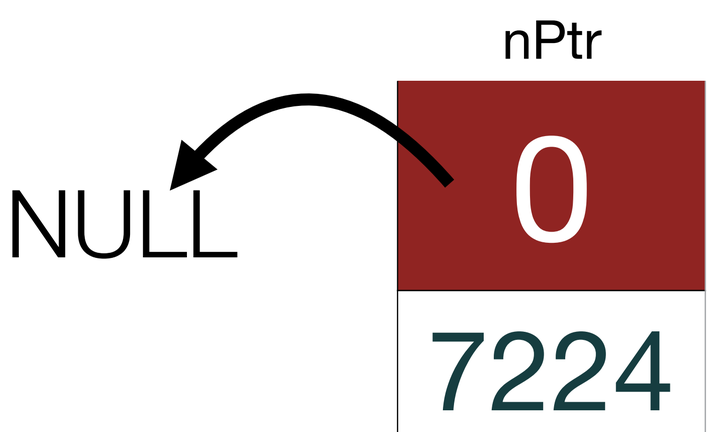
What should we draw?



Pointer Practice

These little boxes we draw to show the memory are so, so important to understanding what is happening. Always draw boxes when learning pointers!

```
int *nPtr = NULL;
int n = 16;
```

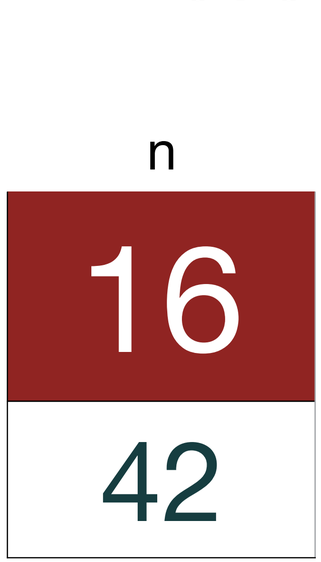


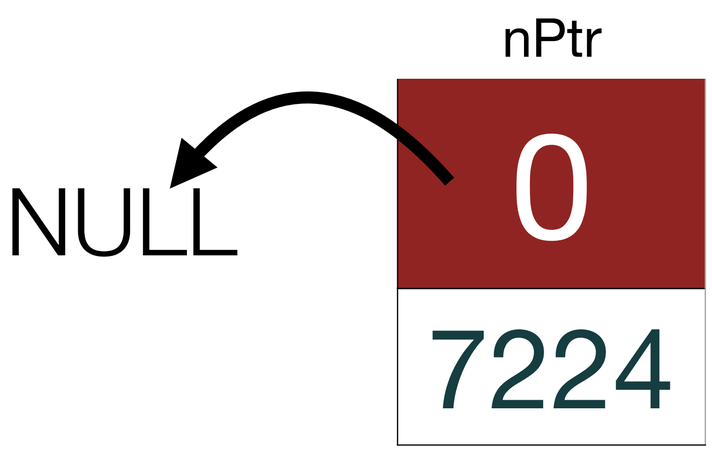
Pointer Practice

These little boxes we draw to show the memory are so, so important to understanding what is happening. Always draw boxes when learning pointers!

```
int *nPtr = NULL;
int n = 16;
nPtr = &n;
```

What should we draw and fill in?

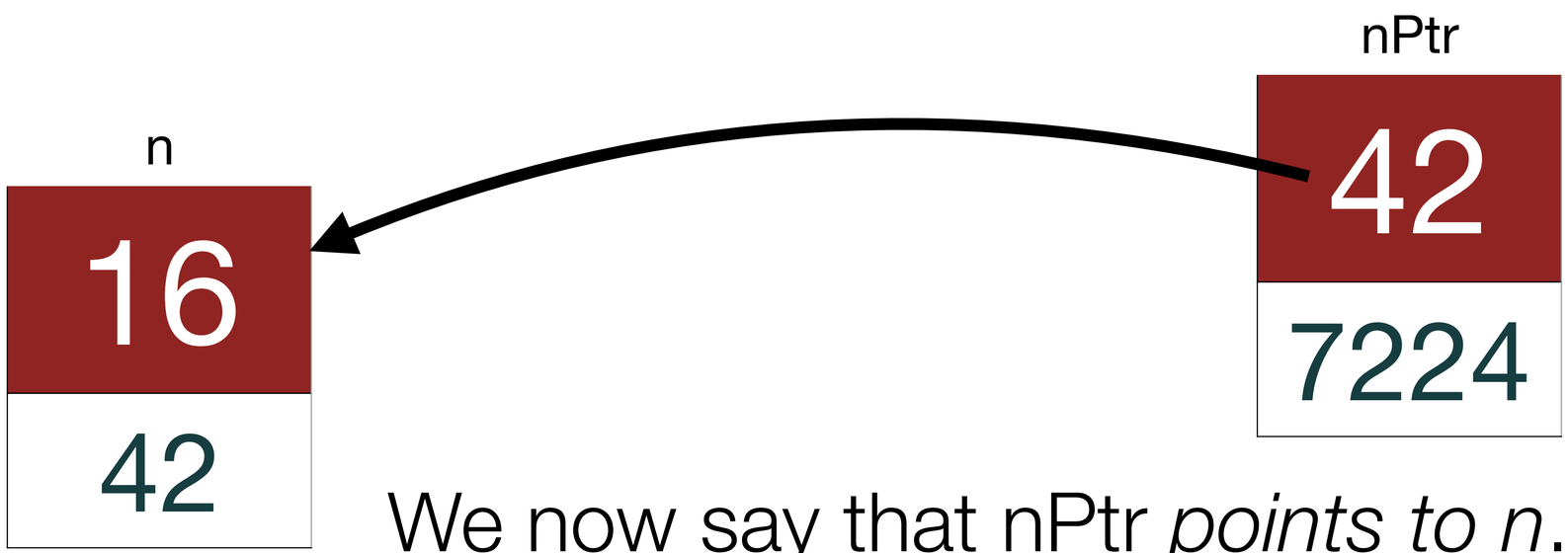




Pointer Practice

These little boxes we draw to show the memory are so, so important to understanding what is happening. Always draw boxes when learning pointers!

```
int *nPtr = NULL;
int n = 16;
nPtr = &n;
```



We now say that nPtr *points to n*.

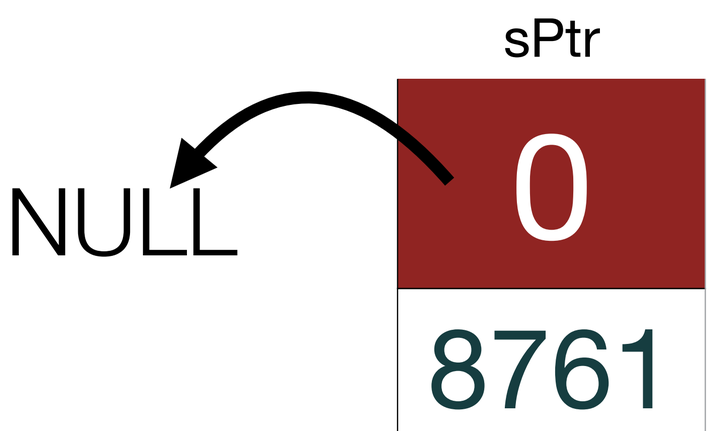
Pointers

What is a pointer??

a memory address!

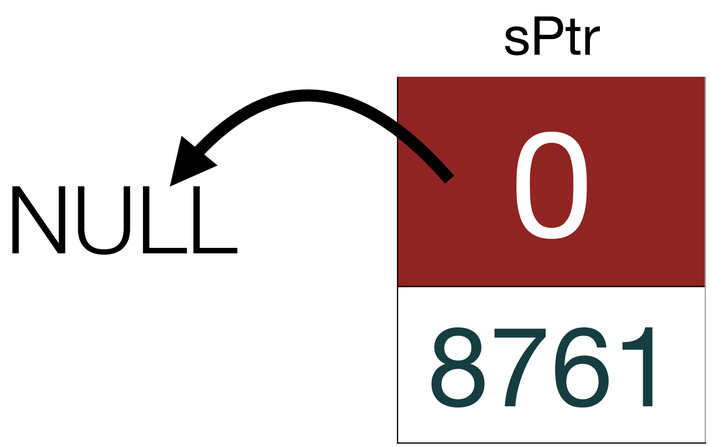
Pointer Practice

```
string *sPtr = NULL;
string s = "hello";
sPtr = &s;
cout << *sPtr << endl;
```



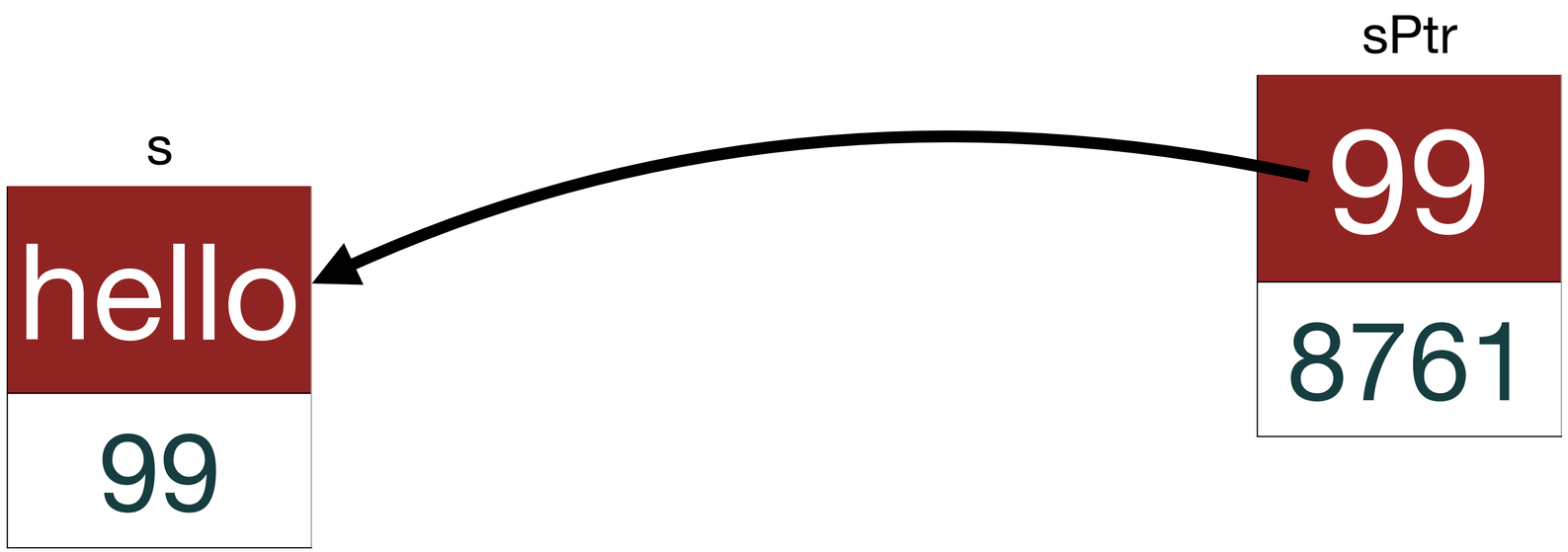
Pointer Practice

```
string *sPtr = NULL;
string s = "hello";
sPtr = &s;
cout << *sPtr << endl;
```



Pointer Practice

```
string *sPtr = NULL;
string s = "hello";
sPtr = &s;
cout << *sPtr << endl;
```

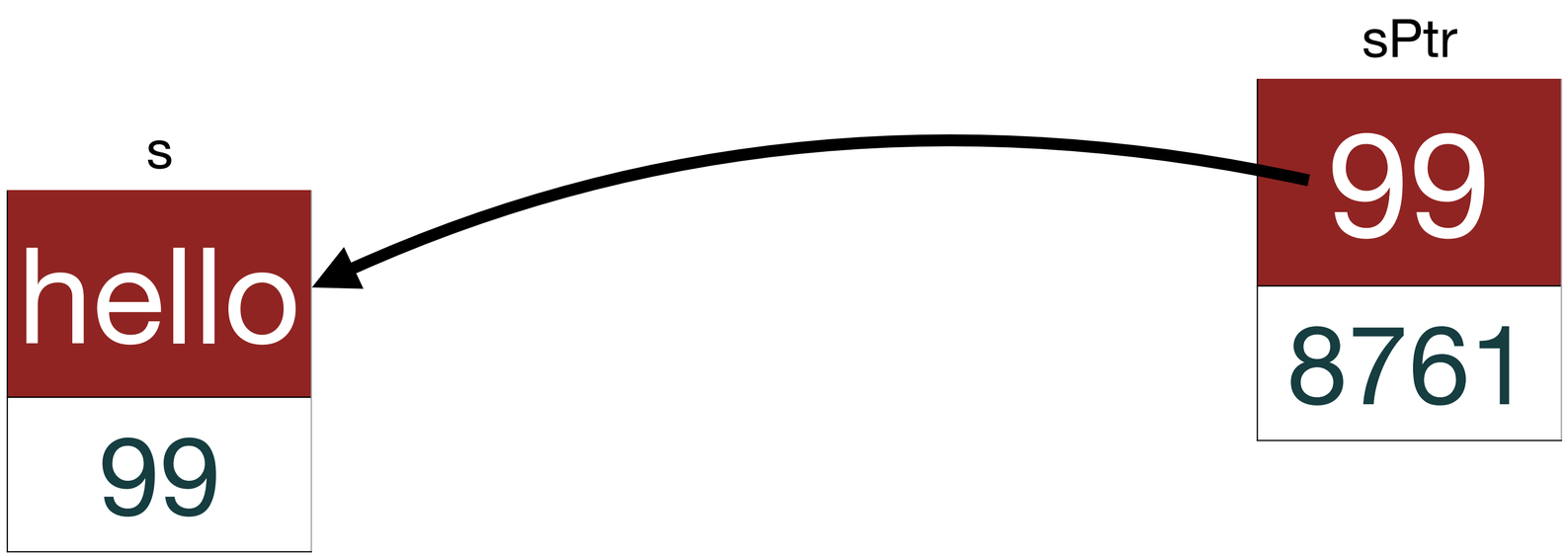


Pointer Practice

```
string *sPtr = NULL;
string s = "hello";
sPtr = &s;
cout << *sPtr << endl;
```

Output:

hello



Pointer Practice

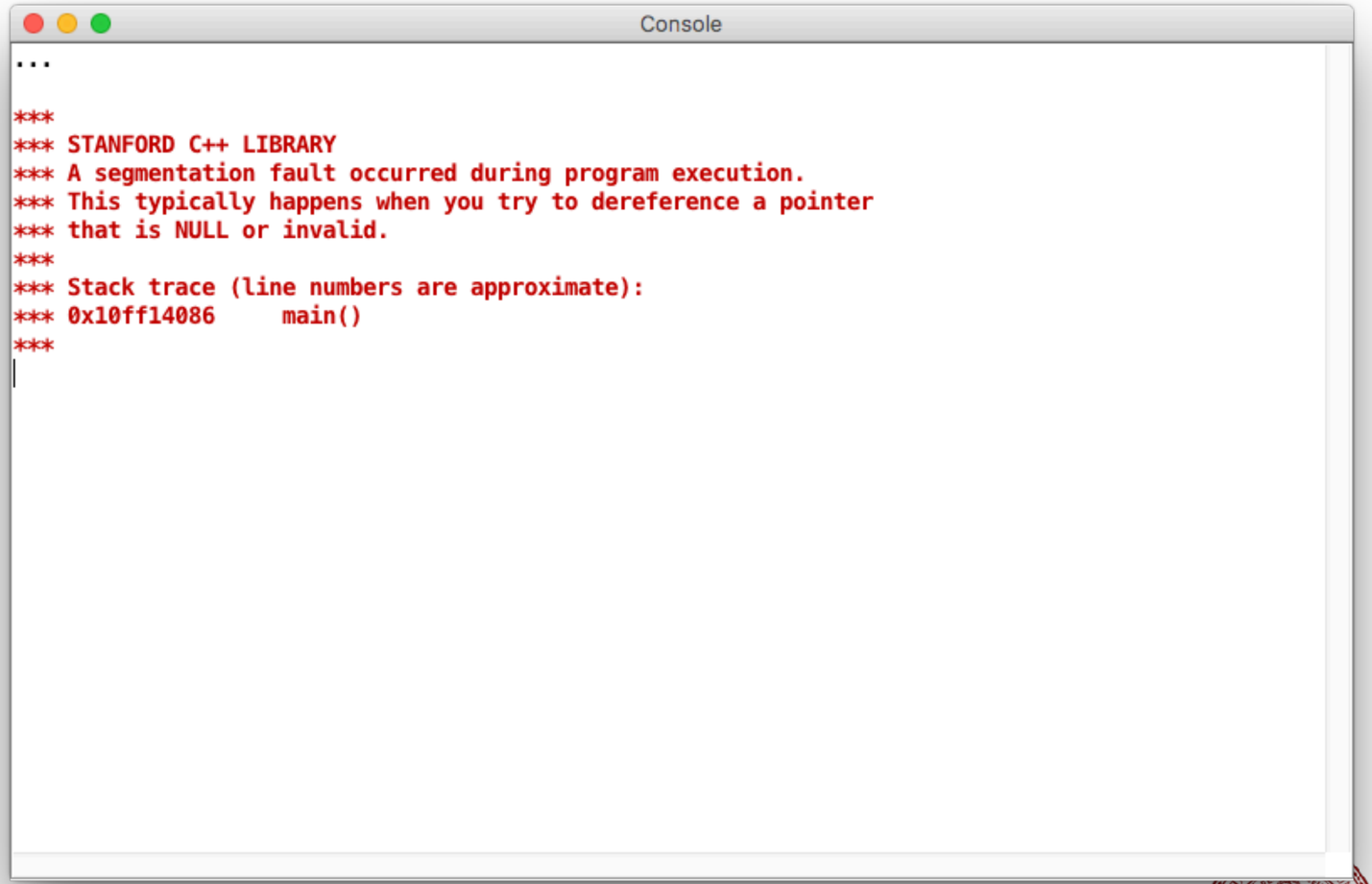
```
string *sPtr = NULL;
string s = "hello";
cout << *sPtr << endl;
```

Output?

Pointer Practice

```
string *sPtr = NULL;
string s = "hello";
cout << *sPtr << endl;
```

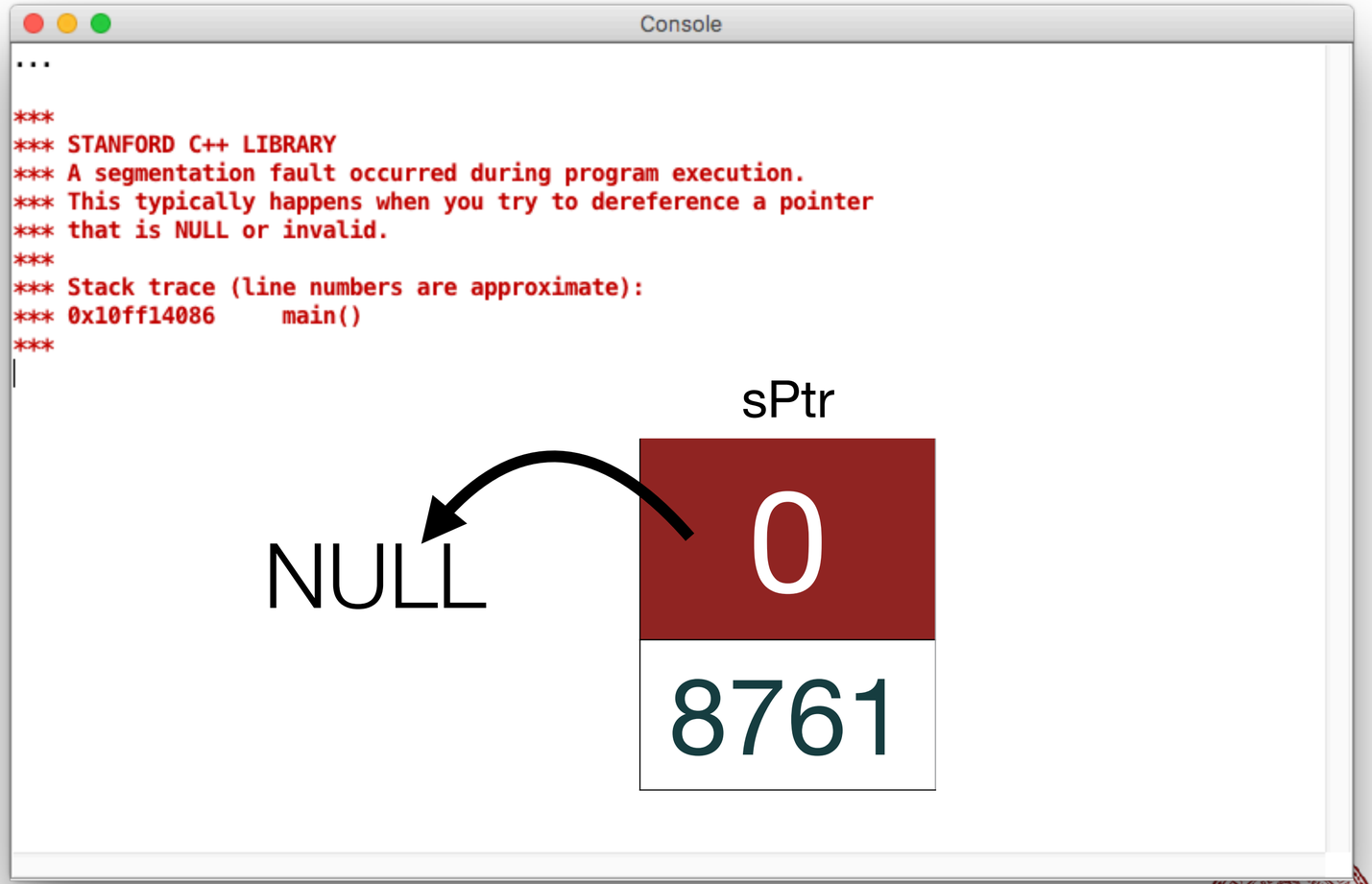
Output? Seg Fault! (crash!)



Pointer Practice

```
string *sPtr = NULL;
string s = "hello";
cout << *sPtr << endl;
```

Output? Seg Fault! (crash!)



Be careful when dereferencing pointers!

Pointer Practice

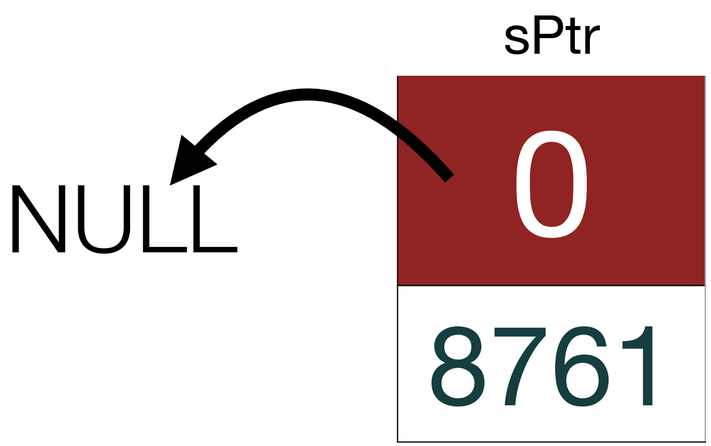
- You can also use the dereferencing operator to *set* the value of the "pointee" (the variable being pointed to):

```
string *sPtr = NULL;
string s = "hello";
sPtr = &s;
*sPtr = "goodbye";
cout << s << endl;
```

Pointer Practice

- You can also use the dereferencing operator to *set* the value of the "pointee" (the variable being pointed to):

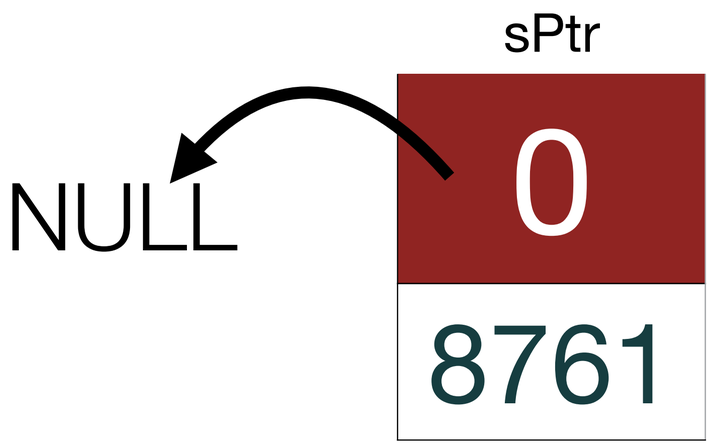
```
string *sPtr = NULL;
string s = "hello";
sPtr = &s;
*sPtr = "goodbye";
cout << s << endl;
```



Pointer Practice

- You can also use the dereferencing operator to *set* the value of the "pointee" (the variable being pointed to):

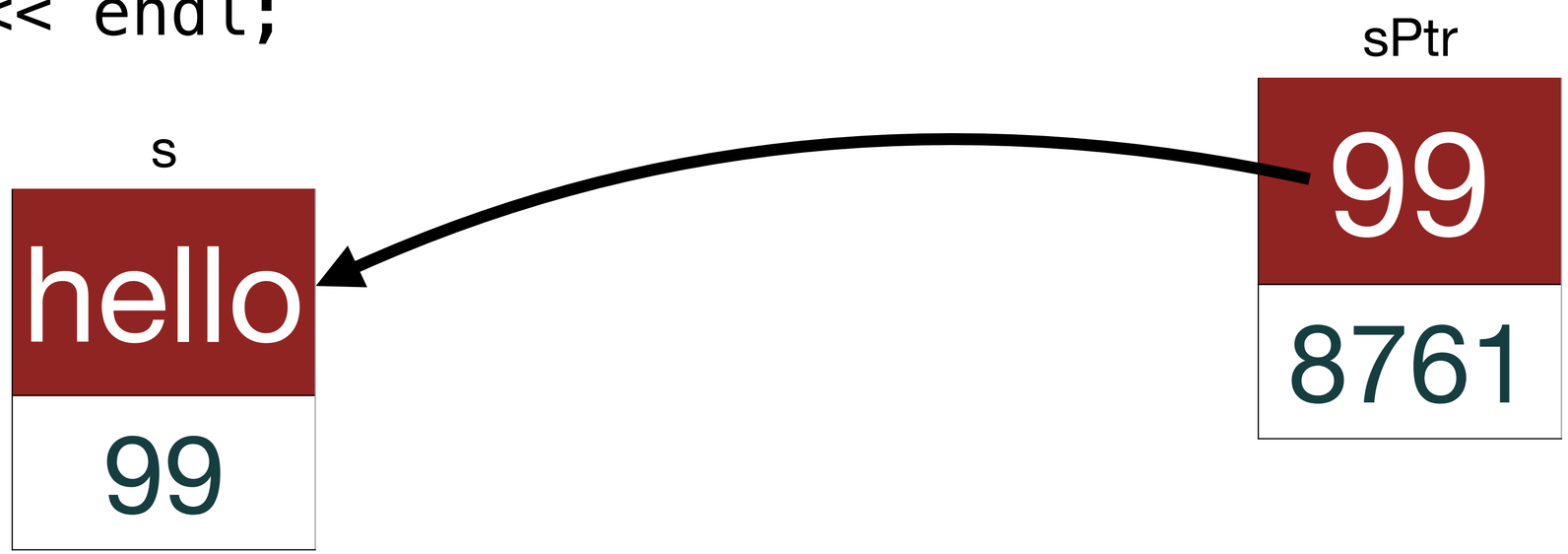
```
string *sPtr = NULL;
string s = "hello";
sPtr = &s;
*sPtr = "goodbye";
cout << s << endl;
```



Pointer Practice

- You can also use the dereferencing operator to *set* the value of the "pointee" (the variable being pointed to):

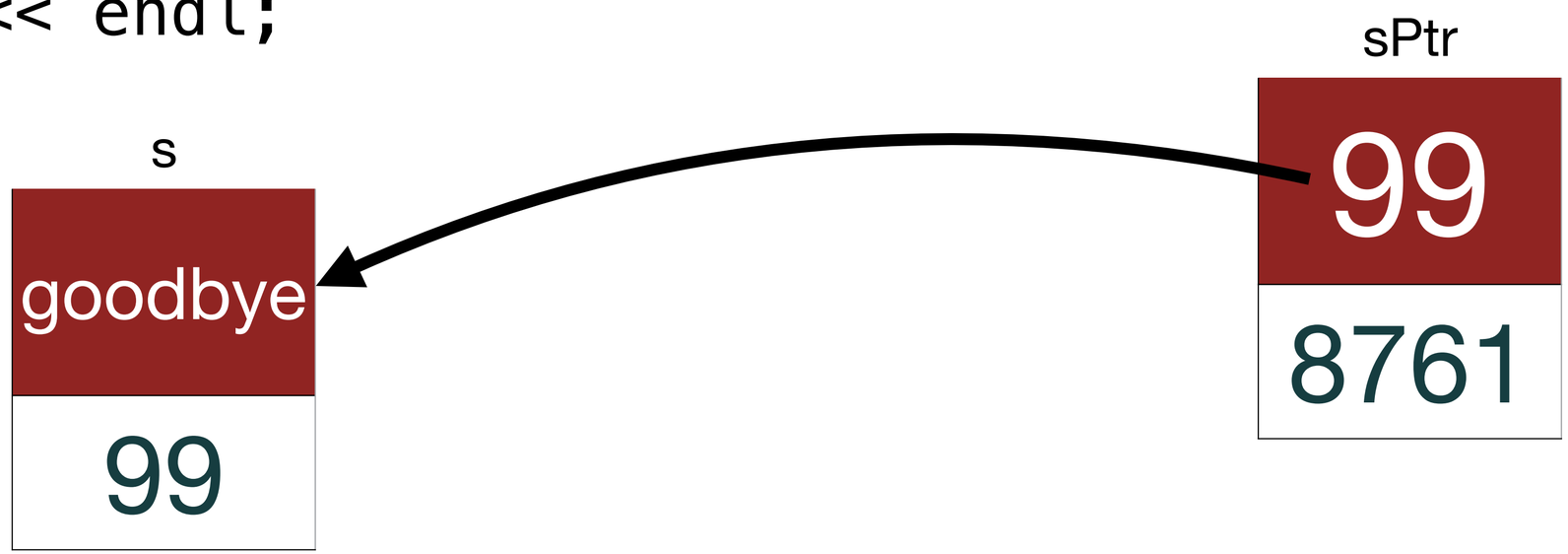
```
string *sPtr = NULL;
string s = "hello";
sPtr = &s;
*sPtr = "goodbye";
cout << s << endl;
```



Pointer Practice

- You can also use the dereferencing operator to *set* the value of the "pointee" (the variable being pointed to):

```
string *sPtr = NULL;
string s = "hello";
sPtr = &s;
*sPtr = "goodbye";
cout << s << endl;
```



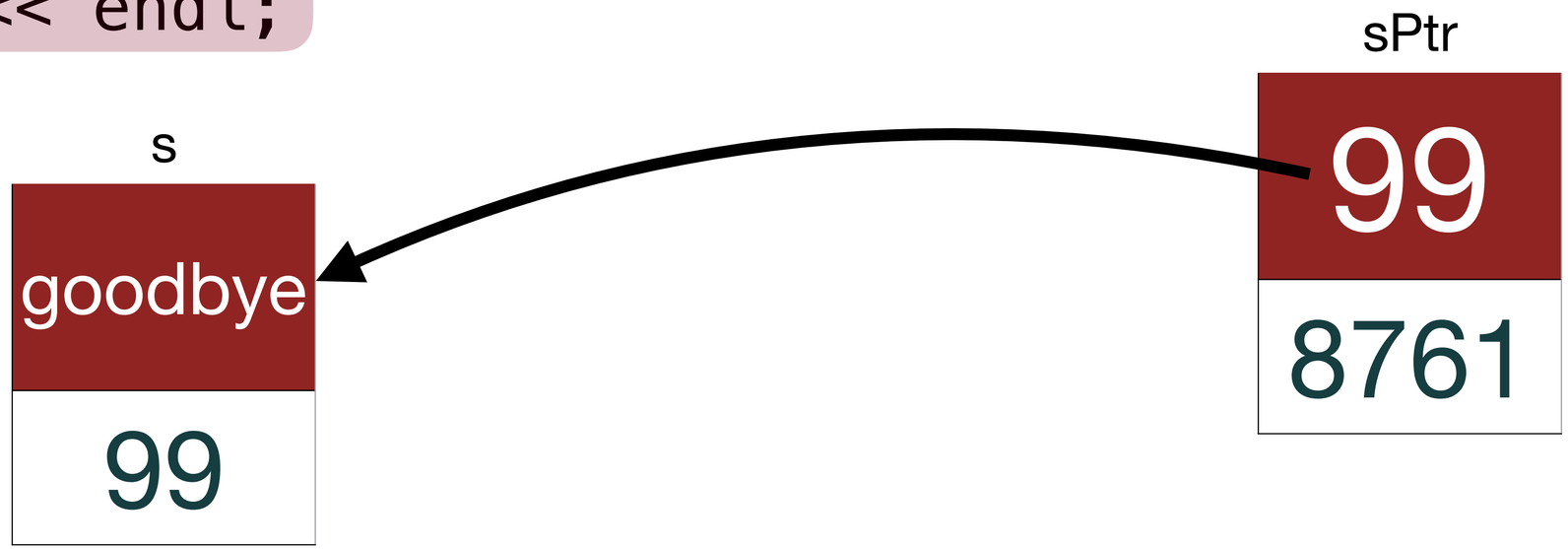
Pointer Practice

- You can also use the dereferencing operator to *set* the value of the "pointee" (the variable being pointed to):

```
string *sPtr = NULL;
string s = "hello";
sPtr = &s;
*sPtr = "goodbye";
cout << s << endl;
```

Output:

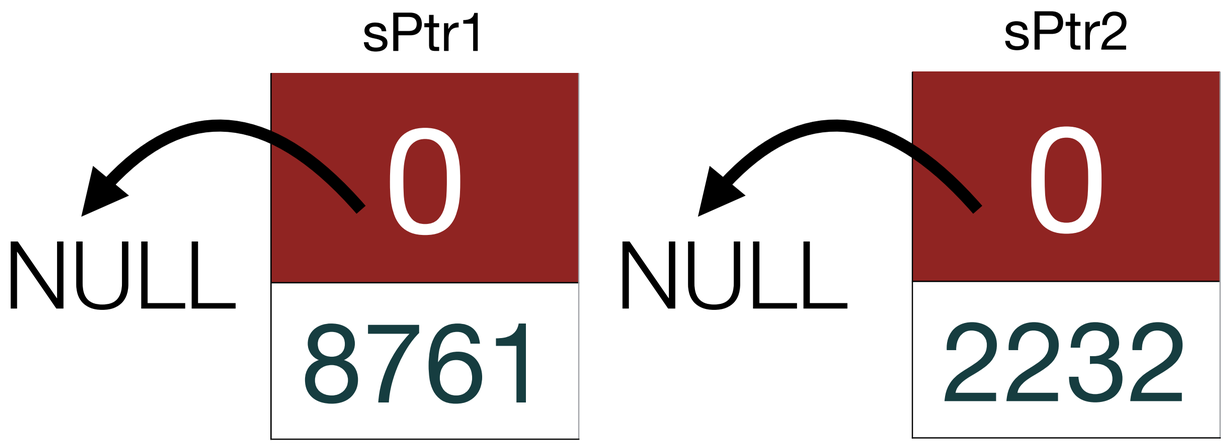
goodbye



Pointer Practice

- If you set one pointer equal to another pointer, they *both point to the same variable!*

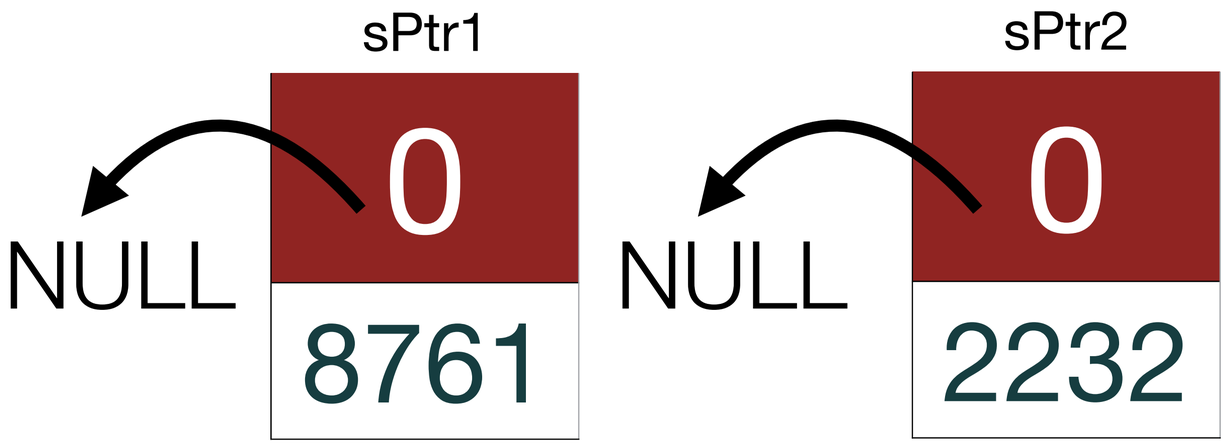
```
string *sPtr1 = NULL;
string *sPtr2 = NULL;
string s = "hello";
sPtr1 = &s;
cout << *sPtr1 << endl;

sPtr2 = sPtr1;
cout << *sPtr2 << endl;
```



Pointer Practice

- If you set one pointer equal to another pointer, they *both point to the same variable!*

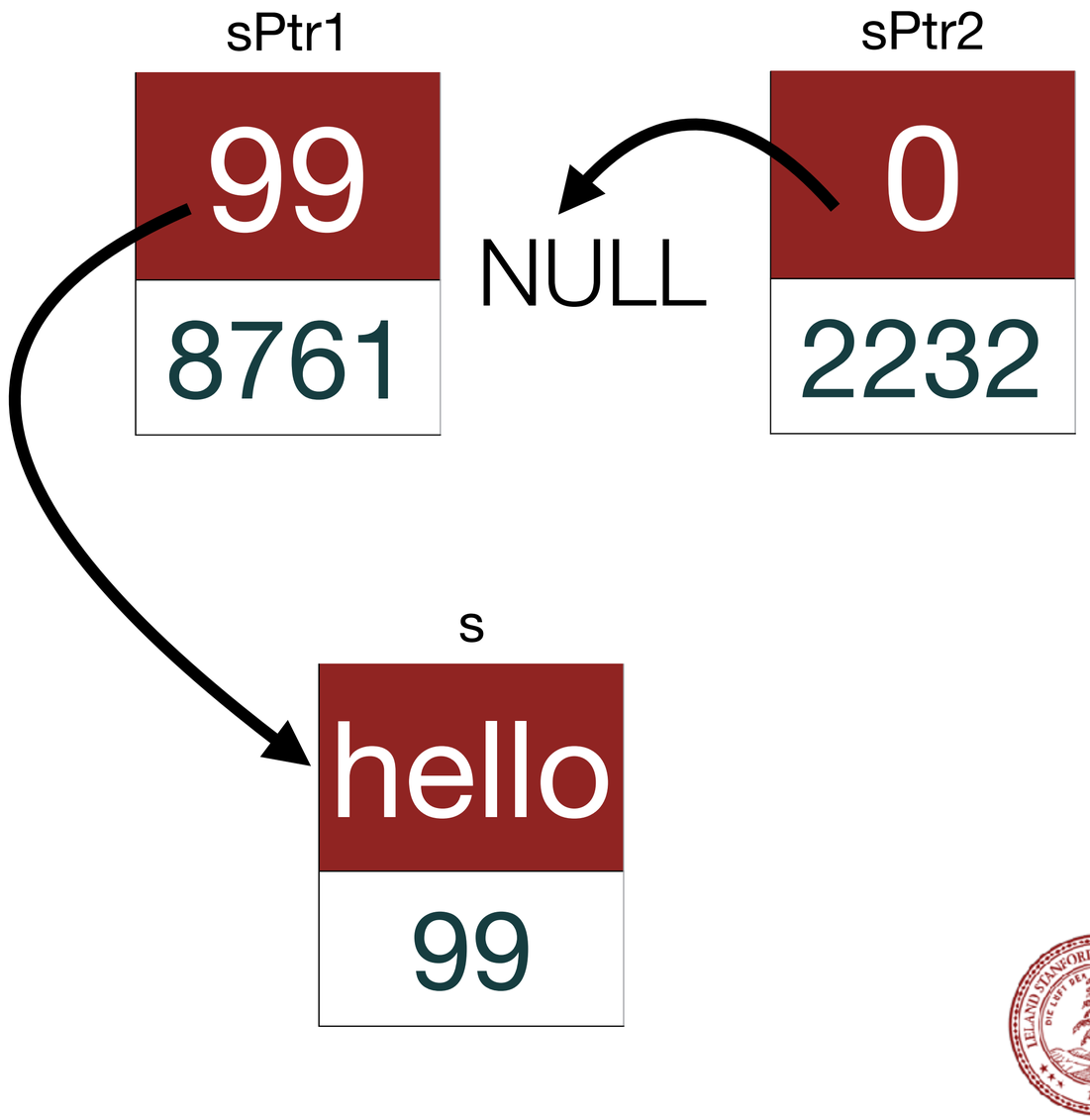
```
string *sPtr1 = NULL;
string *sPtr2 = NULL;
string s = "hello";
sPtr1 = &s;
cout << *sPtr1 << endl;

sPtr2 = sPtr1;
cout << *sPtr2 << endl;
```



Pointer Practice

- If you set one pointer equal to another pointer, they *both point to the same variable!*

```
string *sPtr1 = NULL;
string *sPtr2 = NULL;
string s = "hello";
sPtr1 = &s;
cout << *sPtr1 << endl;

sPtr2 = sPtr1;
cout << *sPtr2 << endl;
```



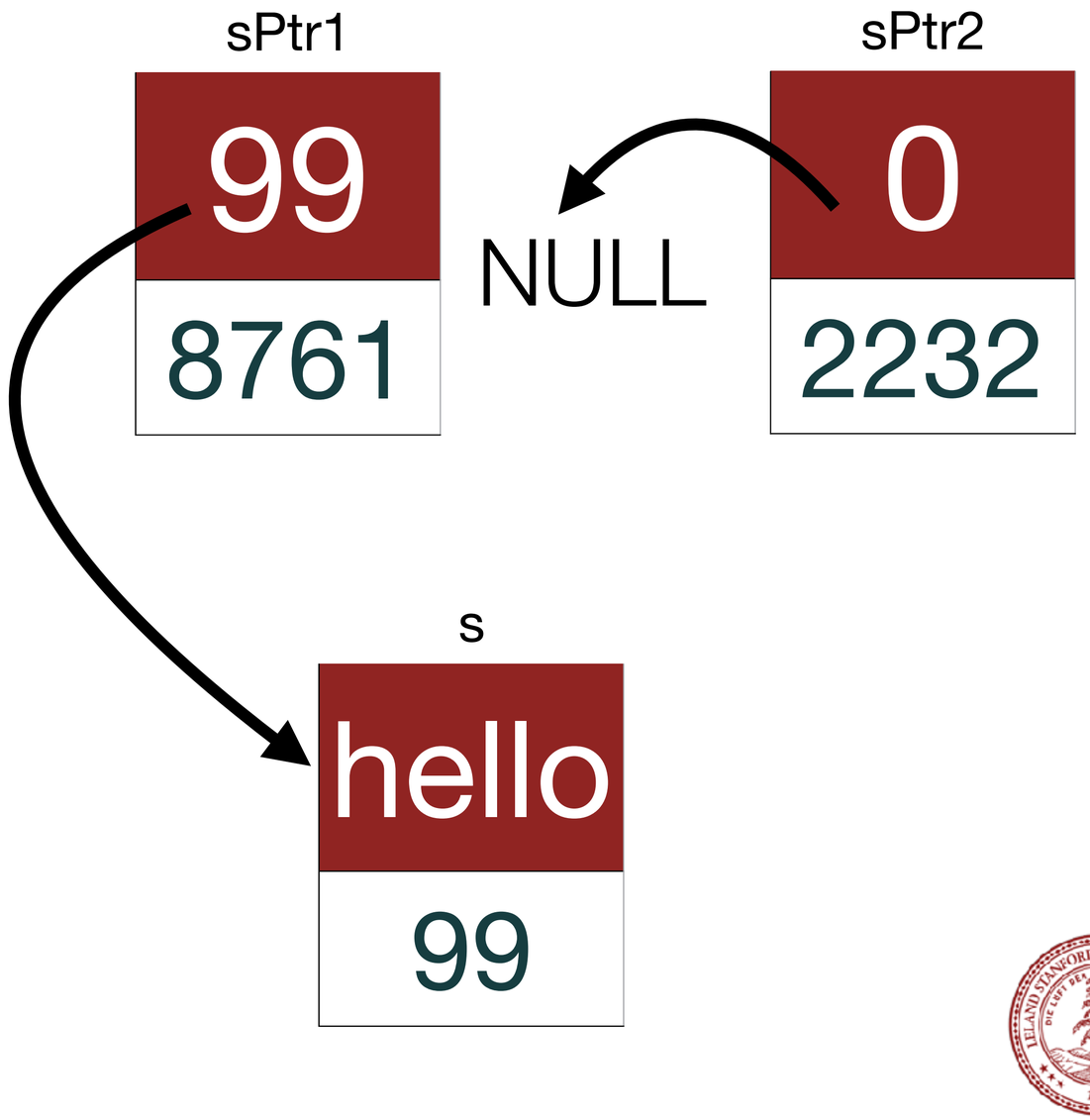
Pointer Practice

- If you set one pointer equal to another pointer, they *both point to the same variable!*

```
string *sPtr1 = NULL;
string *sPtr2 = NULL;
string s = "hello";
sPtr1 = &s;
cout << *sPtr1 << endl;

sPtr2 = sPtr1;
cout << *sPtr2 << endl;
```

Output:

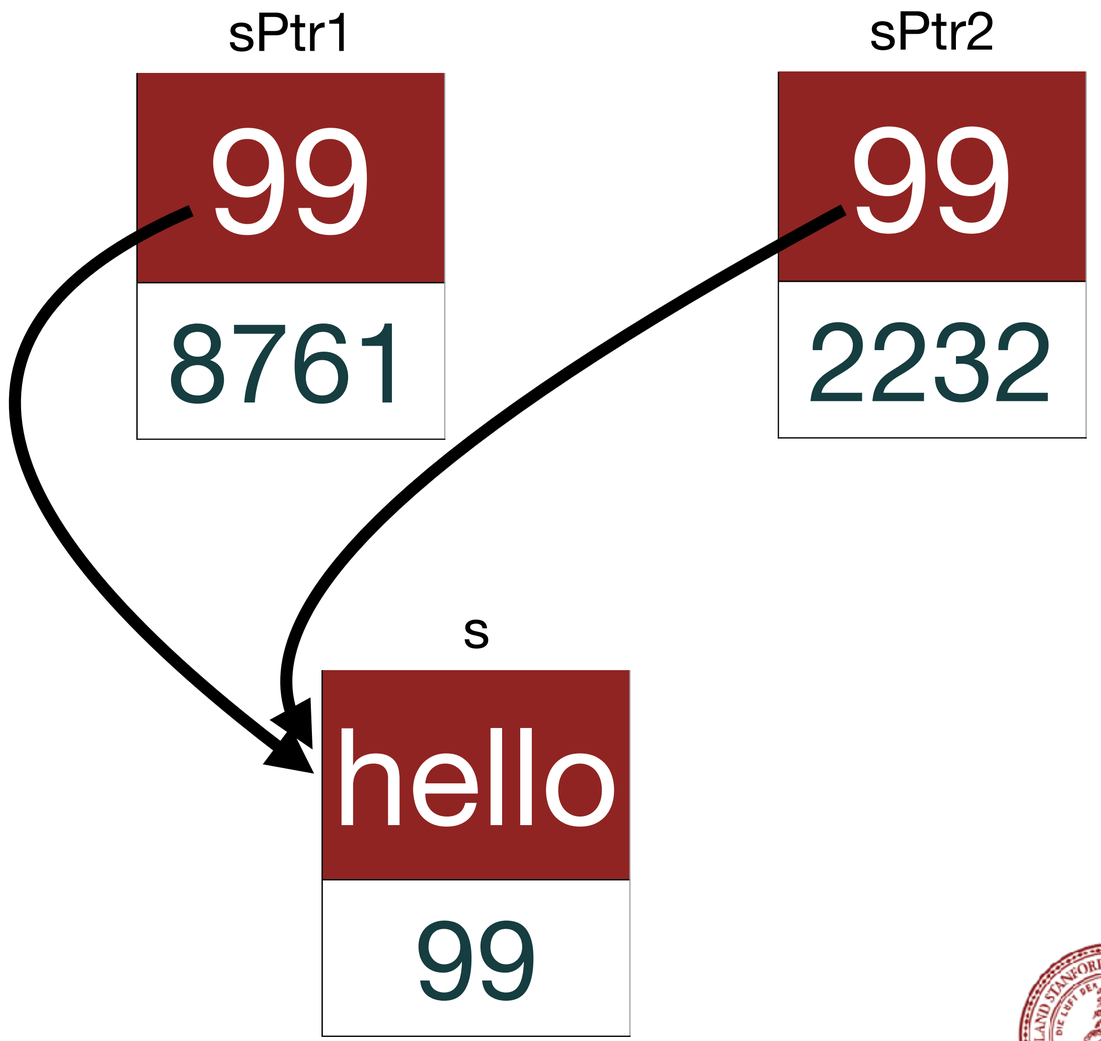
hello



Pointer Practice

- If you set one pointer equal to another pointer, they *both point to the same variable!*

```
string *sPtr1 = NULL;
string *sPtr2 = NULL;
string s = "hello";
sPtr1 = &s;
cout << *sPtr1 << endl;

sPtr2 = sPtr1;
cout << *sPtr2 << endl;
```



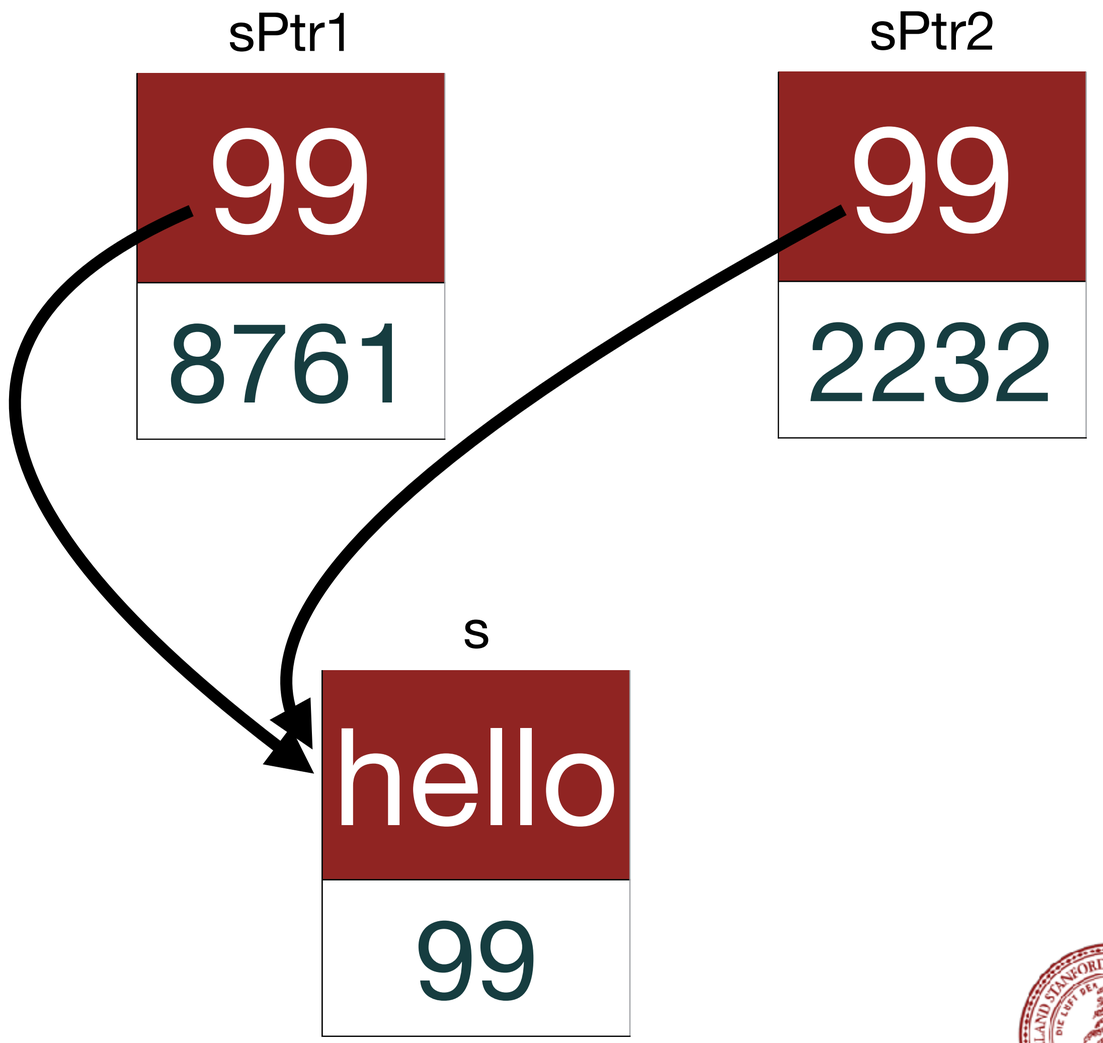
Pointer Practice

- If you set one pointer equal to another pointer, they *both point to the same variable!*

```
string *sPtr1 = NULL;
string *sPtr2 = NULL;
string s = "hello";
sPtr1 = &s;
cout << *sPtr1 << endl;

sPtr2 = sPtr1;
cout << *sPtr2 << endl;
```

Output:

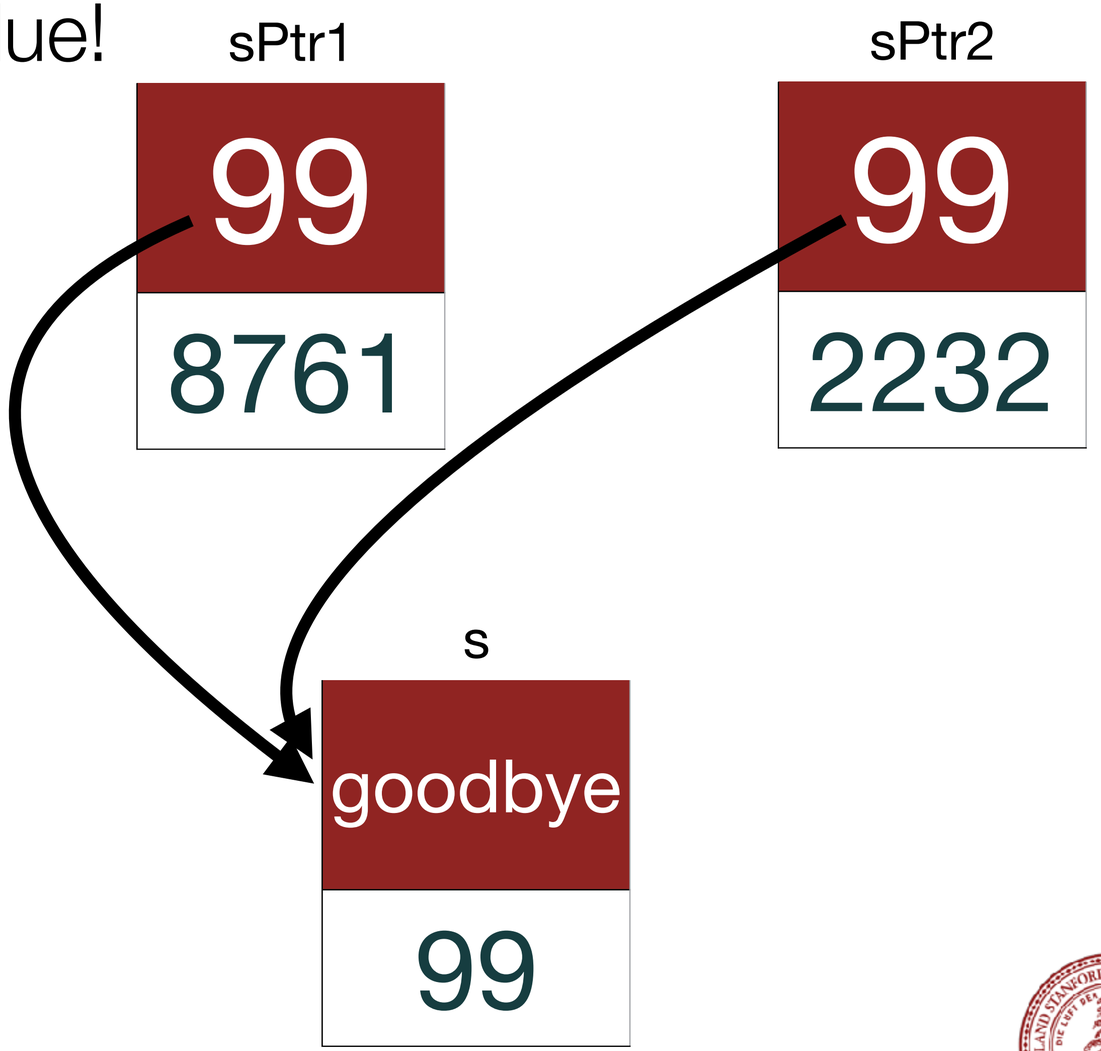
hello



Pointer Practice

- If you dereference and assign a different value, both pointers will now print the same value!

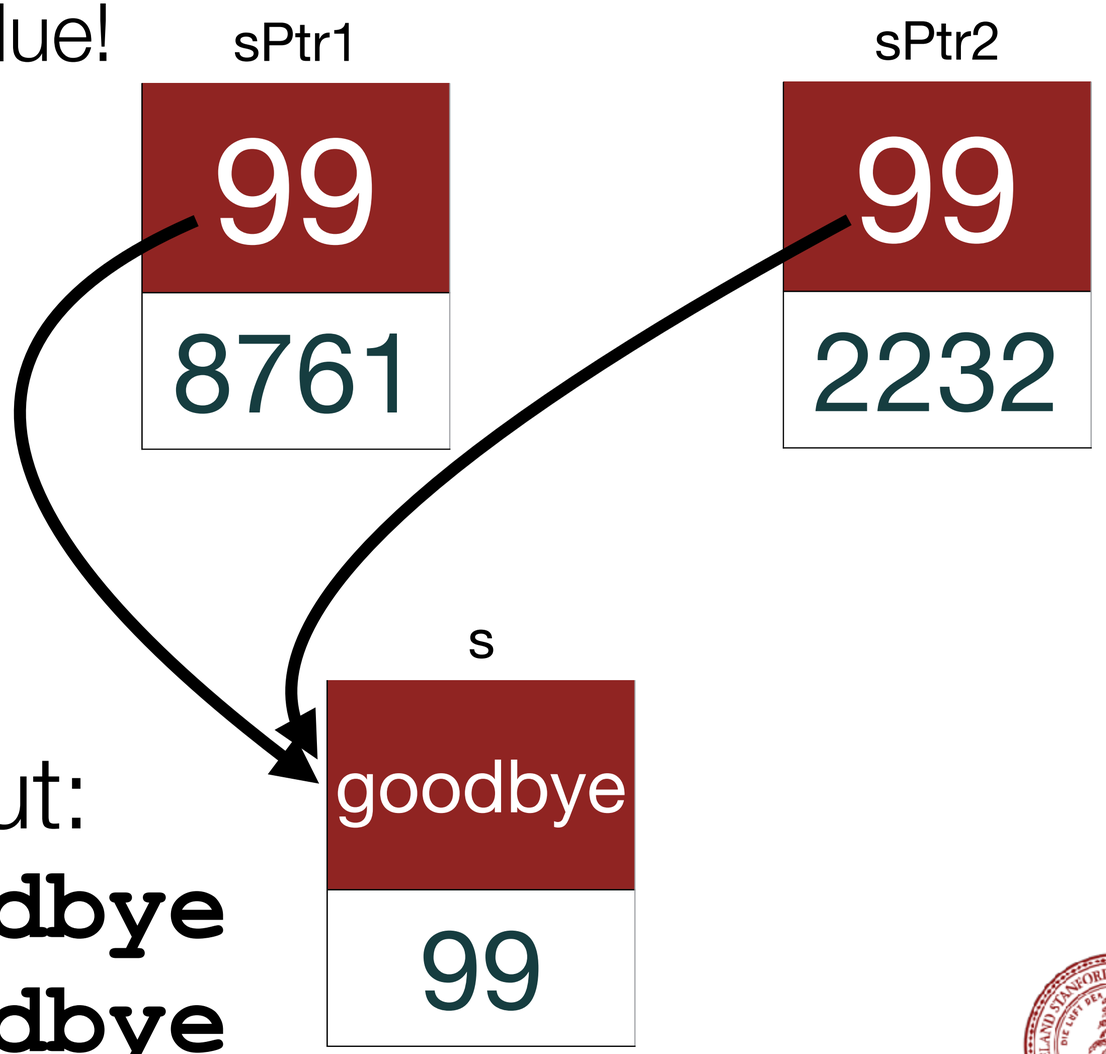
```
string *sPtr1 = NULL;
string *sPtr2 = NULL;
string s = "hello";
sPtr1 = &s;
cout << *sPtr1 << endl;

sPtr2 = sPtr1;
cout << *sPtr2 << endl;

*sPtr1 = "goodbye";
cout << *sPtr1 << endl;
cout << *sPtr2 << endl;
```



Pointer Practice

- If you dereference and assign a different value, both pointers will now print the same value!

```
string *sPtr1 = NULL;
string *sPtr2 = NULL;
string s = "hello";
sPtr1 = &s;
cout << *sPtr1 << endl;

sPtr2 = sPtr1;
cout << *sPtr2 << endl;

*sPtr1 = "goodbye";
cout << *sPtr1 << endl;
cout << *sPtr2 << endl;
```



Output:

goodbye

goodbye

Pointers

What is a pointer??

a memory address!

Pointers

More information about addresses:

Addresses are just numbers, as we have seen. However, you will often see an address listed like this:

`0x7fff3889b4b4`
or this: `0x602a10`

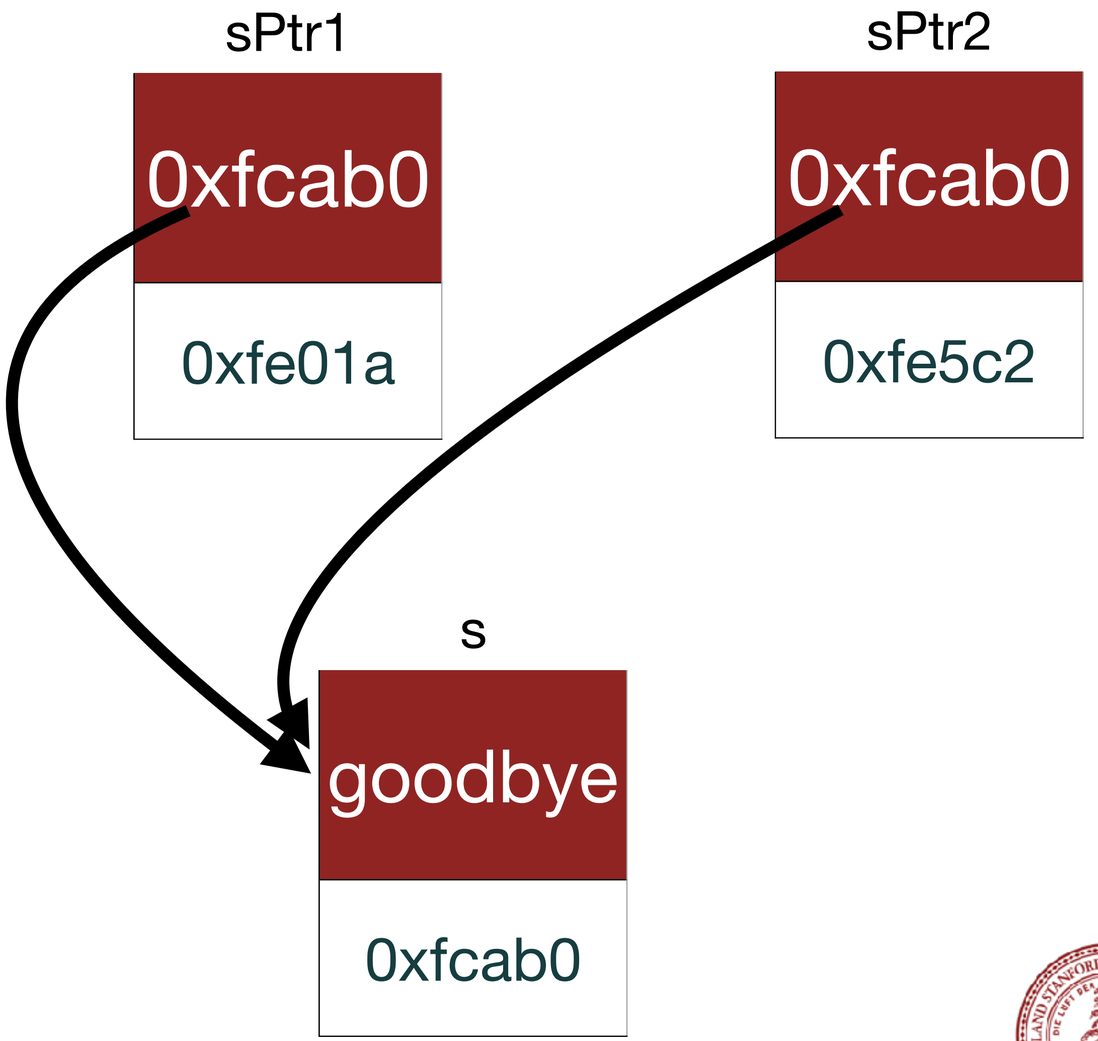
This is a base-16, or "hexadecimal" representation. The **`0x`** just means "the following number is in hexadecimal."

The letters are used because base 16 needs 16 digits:

`0 1 2 3 4 5 6 7 8 9 a b c d e f`

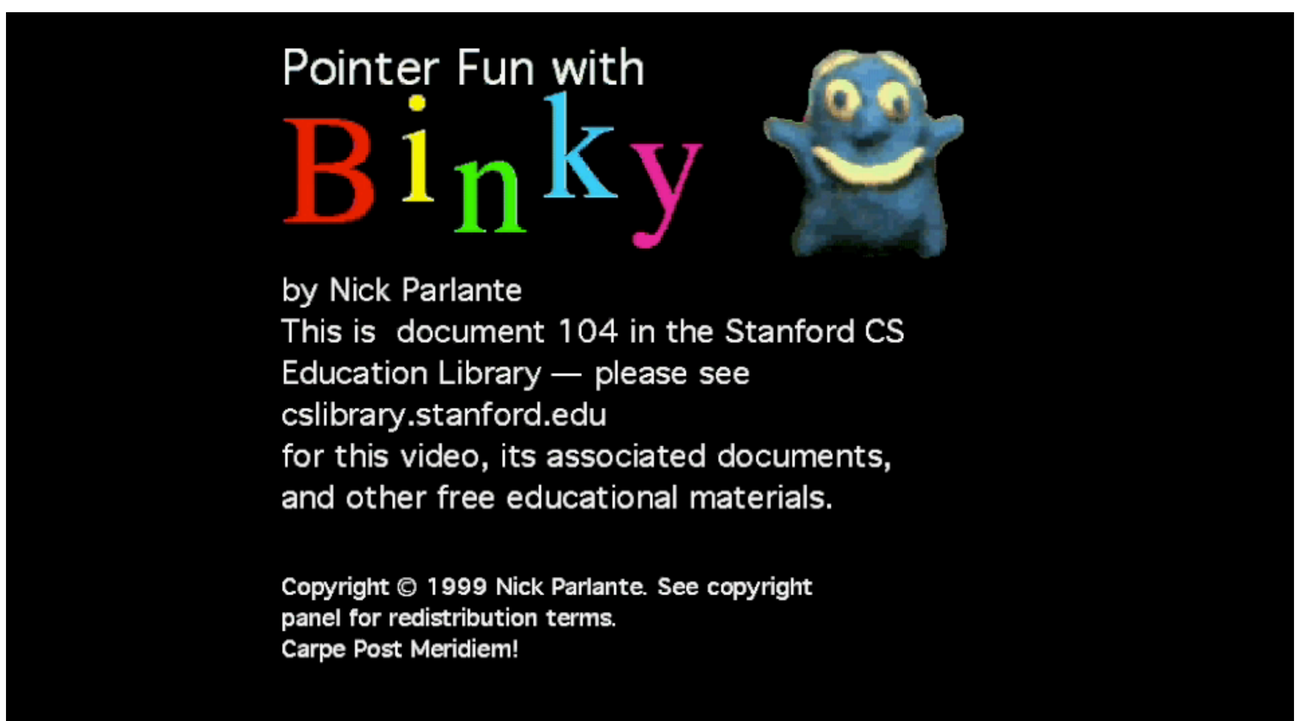
Pointer Practice

- So, you might see the following -- remember, we don't actually care about the address values, just that they are memory locations.



Binky

- Our very own Nick Parlante (another CS Lecturer) put this video together many years ago:

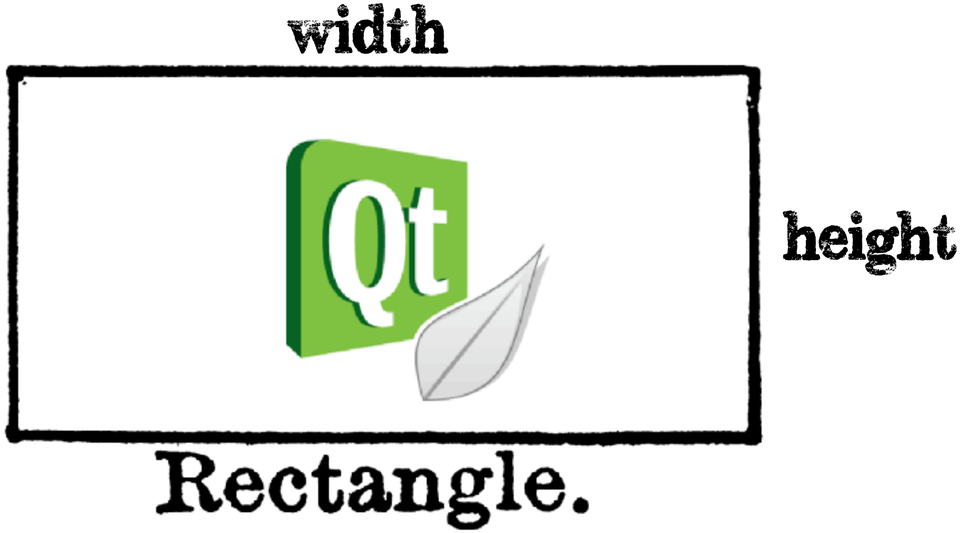


Let's write our swap function with pointers!

Let's write our swap function with pointers!

```
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

Back to Classes: the Copy Constructor



- Okay, now we know about classes and pointers. We will learn about the "new" operator on Monday, but for now just know that it gives us space to hold variables that doesn't get lost when a function ends.

- Let's take a look at a simple Rectangle class (almost certainly **not** the way we would really write this class)

Back to Classes: the Copy Constructor

rectangle.h:

```
#pragma once

class Rectangle {
public:
    Rectangle(double width = 1, double height = 1); // constructor
    ~Rectangle(); // destructor (more on this later)

    double area();
    double perimeter();
    double getHeight();
    double getWidth();

private:
    double *height; // pointer to a double
    double *width;  // pointer to a double
};
```

Back to Classes: the Copy Constructor

rectangle.cpp:

```
#include "rectangle.h"

Rectangle::Rectangle(double width, double height) { // constructor
    this->width = new double;
    this->height = new double;
    *(this->width) = width;
    *(this->width) = height;
}

Rectangle::~Rectangle() { // destructor
    delete height;
    delete width;
}

double Rectangle::area() {
    return *width * *height;
}

double Rectangle::perimeter() {
    return 2 * *width + 2 * *height;
}

double Rectangle::getHeight() {
    return *height;
}

double Rectangle::getWidth() {
    return *width;
}
```

Back to Classes: the Copy Constructor

rectangle.cpp:

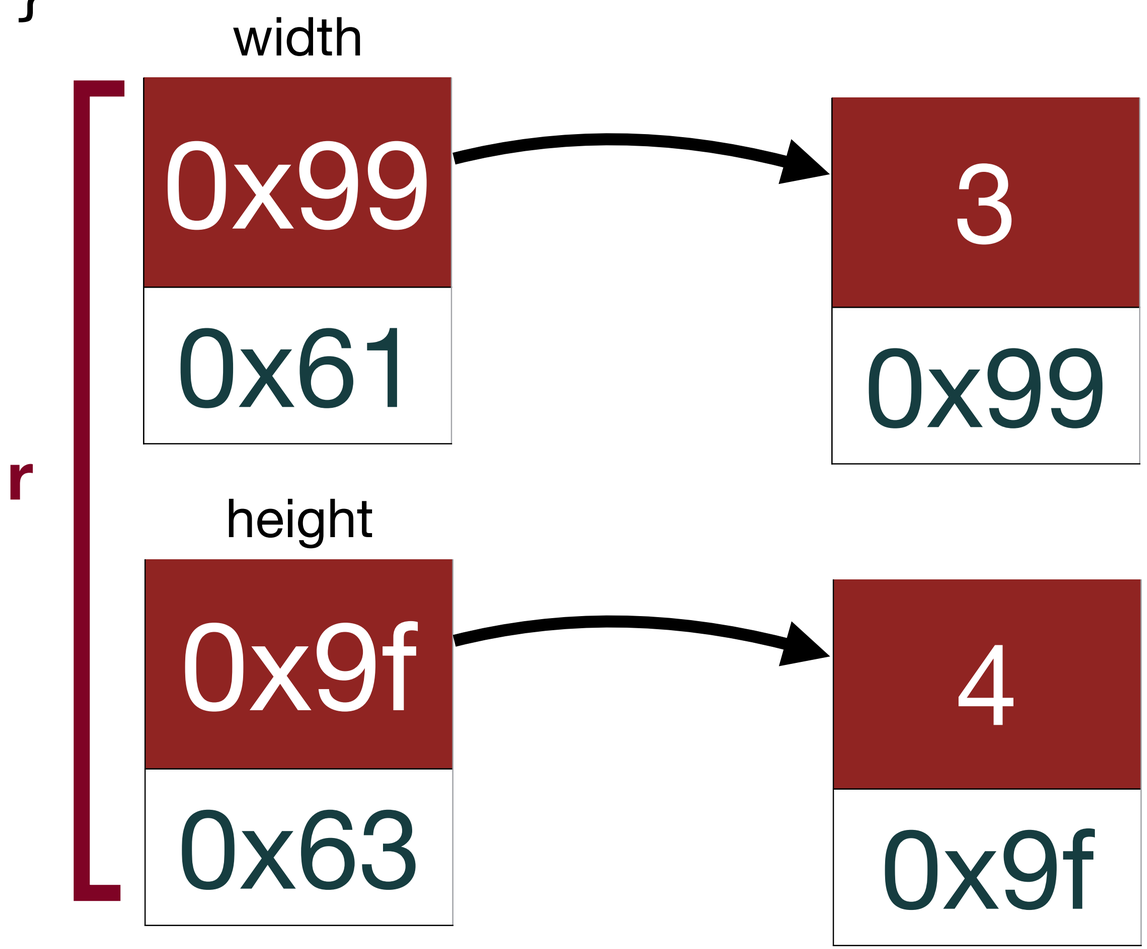
```
int main() {
    Rectangle r(3,4);
    cout << "Width: " << r.getWidth() << ", ";
    cout << "Height: " << r.getHeight() << endl;

    cout << "Area: " << r.area() << endl;
    cout << "Perimeter: " << r.perimeter() << endl;

    // let's make a copy:
    Rectangle r2 = r;

    return 0;
}
```

no problem...

crash!

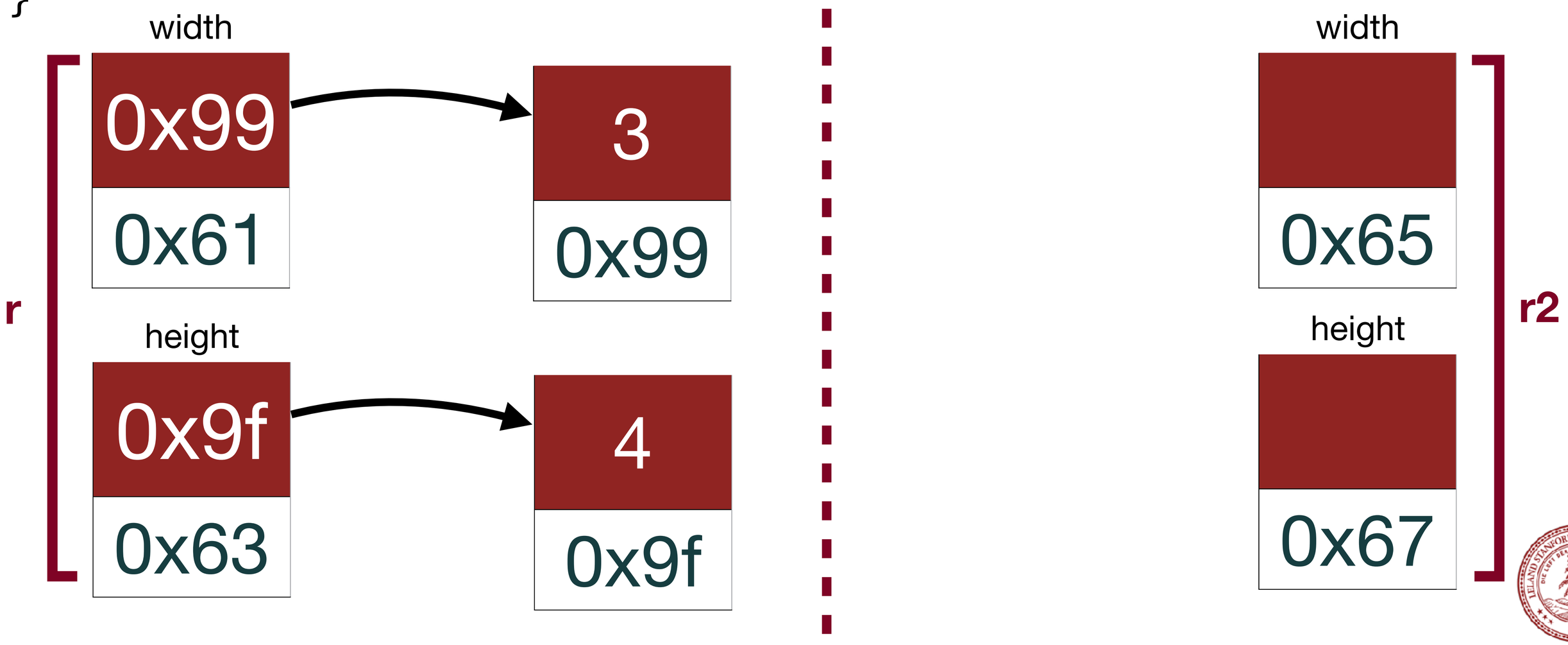
What happened?

```
int main() {
    Rectangle r(3,4);
    Rectangle r2 = r;
}
```



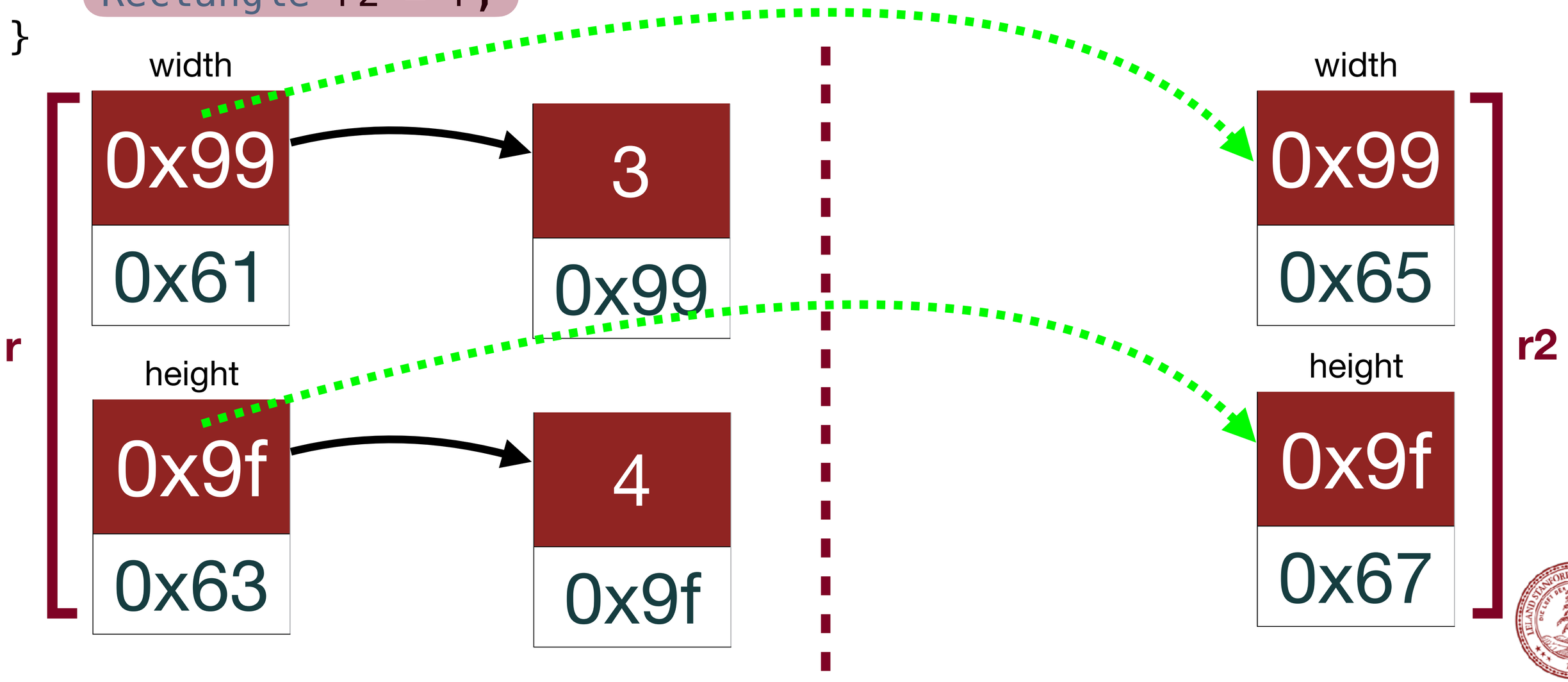
The default is to copy the values...

```
int main() {
    Rectangle r(3,4);
    Rectangle r2 = r;
}
```



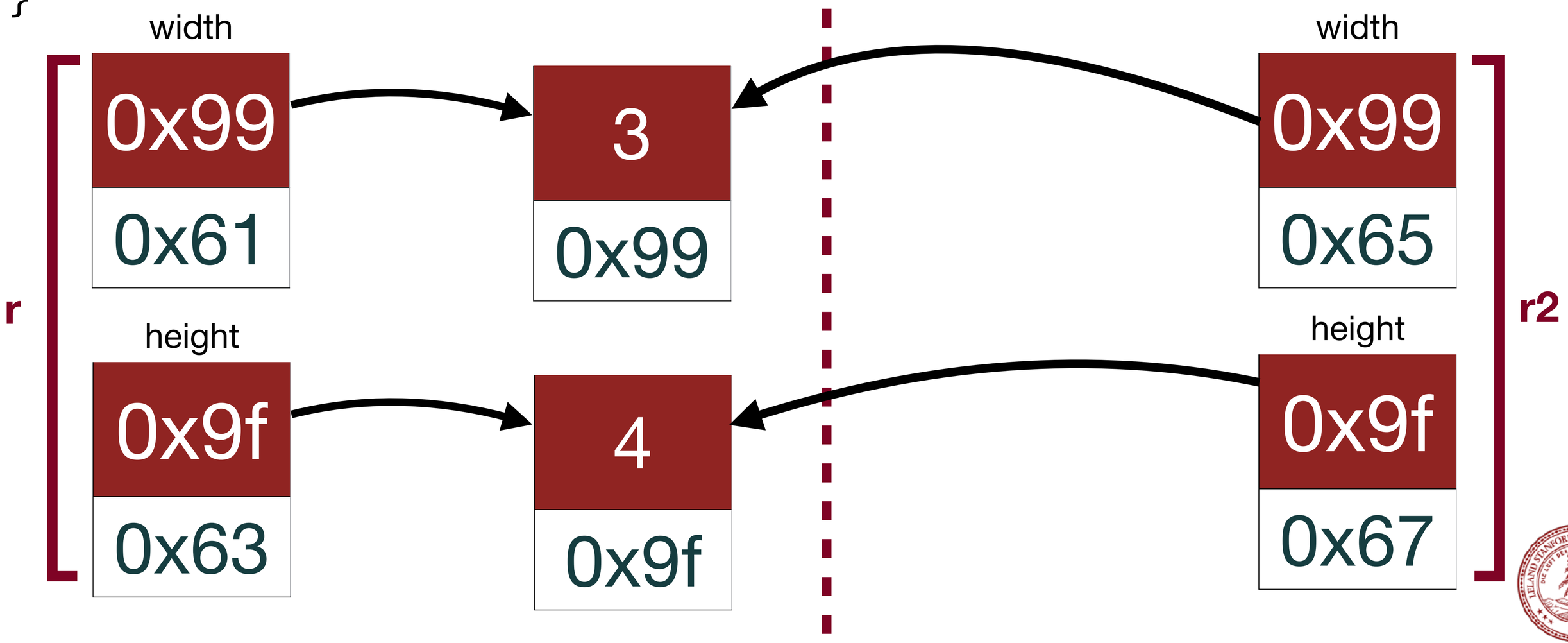
The default is to copy the values...

```
int main() {
    Rectangle r(3,4);
    Rectangle r2 = r;
}
```



Problem! Now both r and r2 point to the same ints!

```
int main() {
    Rectangle r(3,4);
    Rectangle r2 = r;
}
```



What to do? Define a "copy constructor"

The copy constructor tells the compiler how to copy your class. It is important to do this so you don't end up with the situation on the previous slides.

add declaration to rectangle.h

```
class Rectangle {
public:
    Rectangle(double height = 1, double width = 1); // constructor
    Rectangle(const Rectangle &src); // copy constructor
                                    ...
```

add to rectangle.cpp

```
Rectangle::Rectangle(const Rectangle &src) { // copy constructor
    width = new double; // request new memory
    height = new double;

    // copy the values
    *width = *src.width;
    *height = *src.height;
}
```

Recap

- Pointers
  - A pointer is just a memory address that refers to the address of another variable
  - Pointers must point to a particular type (int *, char *, string *, etc.)
  - To declare a pointer, use * (e.g., **`string *stPtr`**)
  - To get the address of a variable to store in a pointer, use &
  - To access the value pointed to by a pointer, use the *
  - Watch out for NULL pointers!
  - Two pointers can point to the same variable.

For Next Time

Dynamic Memory Allocation!

new

delete

arrays

assignment overload (similar in principle to the copy constructor)

References and Advanced Reading

- **References:**
  - More on C++ classes: https://www.tutorialspoint.com/cplusplus/cpp_classes_objects.htm
  - C++ Pointers: https://www.tutorialspoint.com/cplusplus/cpp_pointers.htm

- **Advanced Reading:**
  - Fun video on pointers: https://www.youtube.com/watch?v=B7lVHq-cgeU
  - Hexadecimal numbers: http://www.binaryhexconverter.com/hex-to-decimal-converter
  - Pointer arithmetic: https://www.tutorialspoint.com/cplusplus/cpp_pointer_arithmatic.htm
  - More on pointers: https://www.ntu.edu.sg/home/ehchua/programming/cpp/cp4_PointerReference.html