CS 106B
Lecture 15:
Pointers

Friday, October 28, 2016

Programming Abstractions
Fall 2016
Stanford University
Computer Science Department

Lecturer: Chris Gregg

reading:
Programming Abstractions in C++, Chapter 11

Today's Topics

- Logistics
  - Midterm information:
    - Thursday November 3rd 7:00-9:00pm
    - Last name starts with A-R: Cemex Auditorium (GSB)
    - Last name starts with S-Z: Braun Auditorium (Mudd Chemistry Building)
    - Midterm Review Session: Sunday October 30th at 7:00pm-8:30pm in Bishop Auditorium
    - First practice midterm out, will put out another, as well.

- More on classes
  - Example: the Fraction class
- Introduction to Pointers
  - What are pointers?
  - Pointer Syntax
  - Pointer Tips
  - Pointer Practice

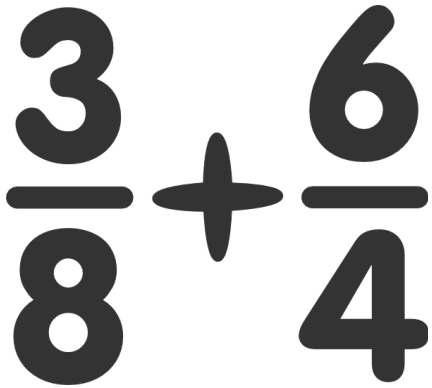
The Fraction Class

- In the last lecture, Chris P. talked about the "Ball" class and the "Bank Account" class.
- As another example of a class, we're going to define a Fraction class that can deal with rational numbers directly, without decimals.
- We are going to walk through the class one step at a time, demonstrating the various parts of a class as we go.

The Fraction Class

- Questions we must answer about the Fraction class:

- What data should the class hold?
- What kinds of functions (public / private) should our class have?
- What constructors could we have?
  - What is a good value for a default fraction?



The Fraction Class

Class outline

```
class Fraction {
public:
    Things we want class users to see
private:
    Things we want to keep hidden
            from class users
};
```

The Fraction Class

Class outline

```
class Fraction {
public:


private:
    int num;   // the numerator
    int denom; // the denominator



};
```

What data would a Fraction class have?

Why is it private?

The Fraction Class

```
class Fraction {
public:
     void add(Fraction f);
     void mult(Fraction f);
     float decimal();
     int getNum();
     int getDenom();
     friend ostream& operator<<
        (ostream& out, Fraction &frac);



private:
     int num; // the numerator
     int denom; // the denominator


};
```

What functions should a fraction class be able to do?

Why are they public?

What is this???

The Fraction Class

```
class Fraction {
public:
     void add(Fraction f);
     void mult(Fraction f);
     float decimal();
     int getNum();
     int getDenom();
     friend ostream& operator<<
        (ostream& out, Fraction &frac);


private:
     int num; // the numerator
     int denom; // the denominator


};
```

What is this???

This defines an operator "overload" to make it possible to use the "<<" operator with cout.

We will write this function in a few minutes.

The Fraction Class

```
class Fraction {
public:
    Fraction();
    Fraction(int num,int denom);
    void add(Fraction f);
    void mult(Fraction f);
    float decimal();
    int getNum();
    int getDenom();
    friend ostream& operator<<
       (ostream& out, Fraction &frac);

private:
    int num; // the numerator
    int denom; // the denominator


};
```

We need to *construct* the class when it is called.

What should a "default" fraction look like?

1 / 1 probably makes the most sense (why not 0/0?)

Should we let the user create an initial fraction, e.g., 3/4?

The Fraction Class

```
class Fraction {
public:
    Fraction();
    Fraction(int num,int denom);
    void add(Fraction f);
    void mult(Fraction f);
    float decimal();
    int getNum();
    int getDenom();
    friend ostream& operator<<
       (ostream& out, Fraction &frac);

private:
    int num; // the numerator
    int denom; // the denominator
    void reduce(); // reduce the fraction
    int gcd(int u, int v);
};
```

Any other functions?

What about reduce? (necessary for multiplication)

Reduce needs gcd()...

The Fraction Class

```
#pragma once
#include<ostream>
using namespace std;

class Fraction {
public:
    Fraction();
    Fraction(int num,int denom);
    void add(Fraction f);
    void mult(Fraction f);
    float decimal();
    int getNum();
    int getDenom();
    friend ostream& operator<<
        (ostream& out, Fraction &frac);
private:
    int num;   // the numerator
    int denom; // the denominator
    void reduce(); // reduce the fraction
    int gcd(int u, int v);
};
```

Last, but not least...

The Fraction Class

Let's start writing our functions. We do this in our fraction.cpp file, and we have to define the class that each function belongs to. We also cannot forget to include our header file!

#include "fraction.h"

The **default constructor** is used when someone wants to just create a default fraction:

```
Fraction frac;
```

```
// purpose: the default constructor
// to create a fraction of 1 / 1
// arguments: none
// return value: none
// (constructors don't return anything)

Fraction::Fraction()
{



}
```

The Fraction Class

Let's start writing our functions. We do this in our fraction.cpp file, and we have to define the class that each function belongs to. We also cannot forget to include our header file!

```
#include "fraction.h"
```

The **default constructor** is used when someone wants to just create a default fraction:

```
Fraction frac;
```

```
// purpose: the default constructor
// to create a fraction of 1 / 1
// arguments: none
// return value: none
// (constructors don't return anything)

Fraction::Fraction()
{

    num   = 1;
    denom = 1;

}
```

Pretty simple! We are just setting our two class variables to default values.

The Fraction Class

We also have an *overloaded* constructor that takes in two values that the user sets. It is called as follows:

```
// create a
// 1/2 fraction
Fraction fracA(1,2);

// create a
// 4/6 fraction
Fraction fracB(4,6);
```

```
// purpose: an overloaded constructor
//          to create a custom fraction
//         that immediately gets reduced
// arguments: an int numerator
//            and an int denominator
Fraction::Fraction(int n,int d)
{









}
```

The Fraction Class

We also have an *overloaded* constructor that takes in two values that the user sets. It is called as follows:

```
// create a
// 1/2 fraction
Fraction fracA(1,2);

// create a
// 4/6 fraction
Fraction fracB(4,6);
```

```
// purpose: an overloaded constructor
//          to create a custom fraction
//         that immediately gets reduced
// arguments: an int numerator
//            and an int denominator
Fraction::Fraction(int n, int d)
{

   num = n;
   denom = d;

  // reduce in case we were given
  // an unreduced fraction
  reduce();


}
```

The Fraction Class

Let's write some more functions...

```
// create two fractions
Fraction fracA(1,2);
Fraction fracB(2,3);

fracA.mult(fracB);
// fracA now holds 1/3
```

```
// purpose: to multiply another fraction
// with this one with the result being
// stored in this fraction
// arguments: another Fraction
// return value: none
void Fraction::mult(Fraction other)
{




}
```

The Fraction Class

Let's write some more functions...

```
// create two fractions
Fraction fracA(1,2);
Fraction fracB(2,3);

fracA.mult(fracB);
// fracA now holds 1/3
```

```
// purpose: to multiply another fraction
// with this one with the result being
// stored in this fraction
// arguments: another Fraction
// return value: none
void Fraction::mult(Fraction other)
{
        // multiplies a Fraction
        // with this Fraction
        num *= other.num;
        denom *= other.denom;

        // reduce the fraction
        reduce();
}
```

The Fraction Class

Let's write some more functions...

```
// get the decimal value
Fraction fracA(1,2);
float f =
fracA.decimal();
cout << f << endl;

output:
0.5
```

```
// purpose: To return a decimal
// value of our fraction
// arguments: None
// return value: the decimal
//                value of this fraction
float Fraction::decimal()
{



}
```

The Fraction Class

Let's write some more functions...

```
// get the decimal value
Fraction fracA(1,2);
float f =
fracA.decimal();
cout << f << endl;

output:
0.5
```

```
// purpose: To return a decimal
// value of our fraction
// arguments: None
// return value: the decimal
//                value of this fraction
float Fraction::decimal()
{
        // returns the decimal
        // value of our fraction
        return (float)num / denom;
}
```

The Fraction Class: reduce()

```
void Fraction::reduce() {
    // reduce the fraction to lowest terms
    // find the greatest common divisor
    int frac_gcd = gcd(num,denom);

    // reduce by dividing num and denom
    // by the gcd
    num = num / frac_gcd;
    denom = denom / frac_gcd;
}
```

The Fraction Class: gcd() should look familiar!

```
int Fraction::gcd(int u, int v) {
    if (v != 0) {
        return gcd(v,u%v);
    }
    else {
        return u;
    }
}
```

The Fraction Class: overloading <<

Yes, this syntax is a bit strange.

Basically, we are telling the compiler how to cout our Fraction.
You can do something very similar for Boggle.

```
// purpose: To overload the << operator
// for use with cout
// arguments: a reference to an outstream and the
//            fraction we are using
// return value: a reference to the outstream
ostream& operator<<(ostream& out, Fraction &frac) {
    out << frac.num << "/" << frac.denom;
    return out;
}
```

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.

pet

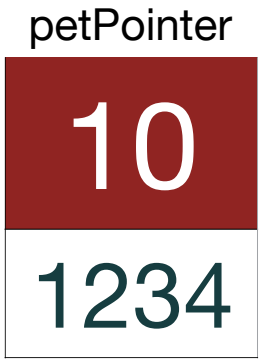
cat
10

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??

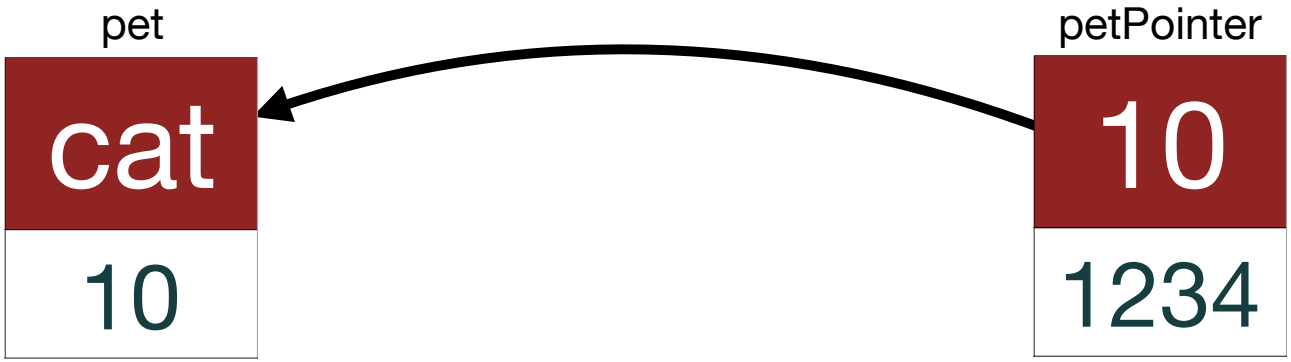
a memory address!

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

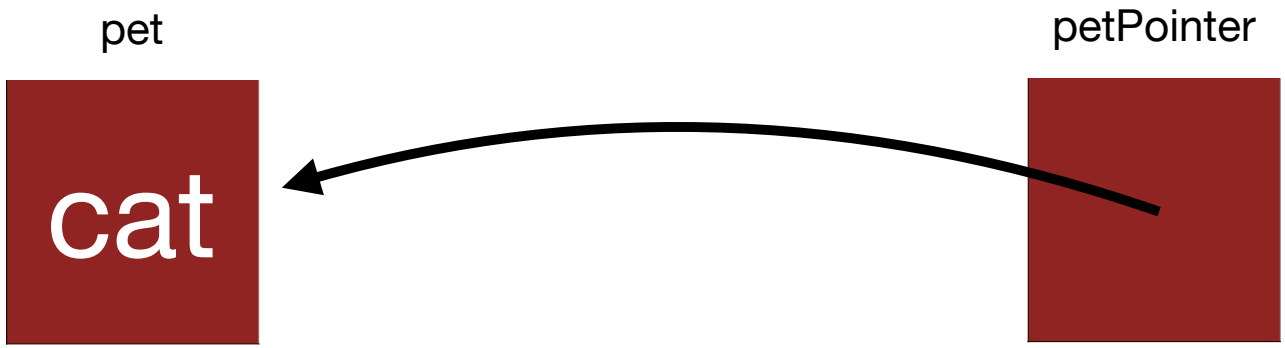


Introduction to Pointers

- 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.

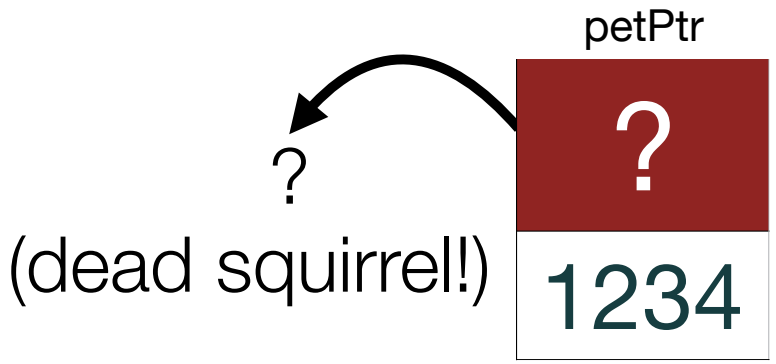
Pointer Syntax

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:

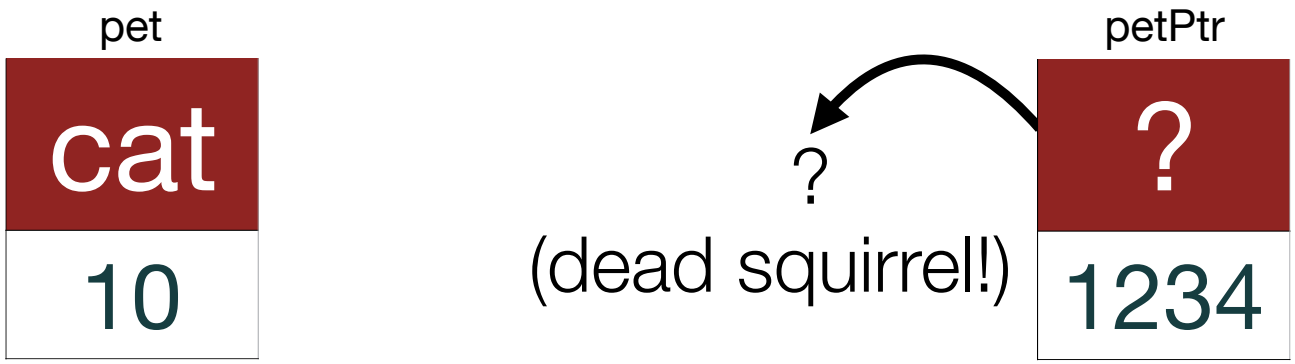
```
string *petPtr; // declare a pointer (which will hold a
                //                    memory address) to a string
```



Pointer Syntax

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

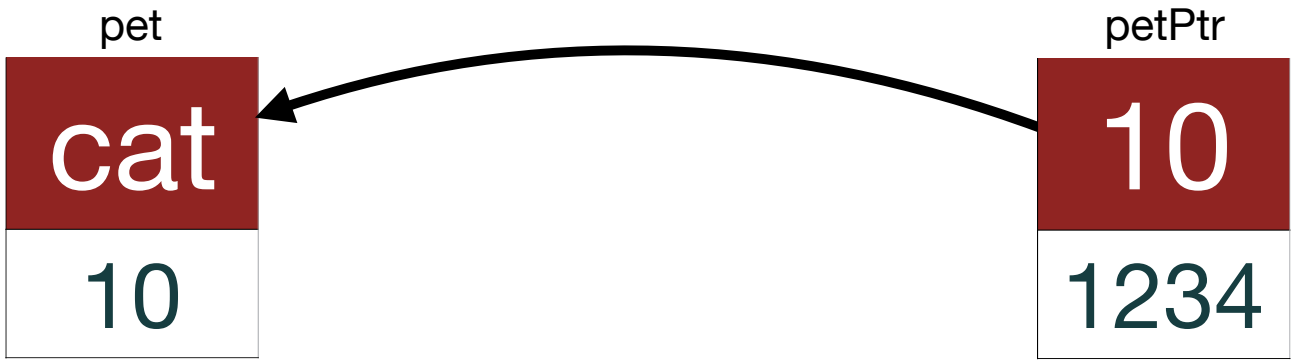
```
string *petPtr; // declare a pointer (which will hold a
                //                    memory address) to a string
string pet = "cat"; // a string variable
```



Pointer Syntax

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

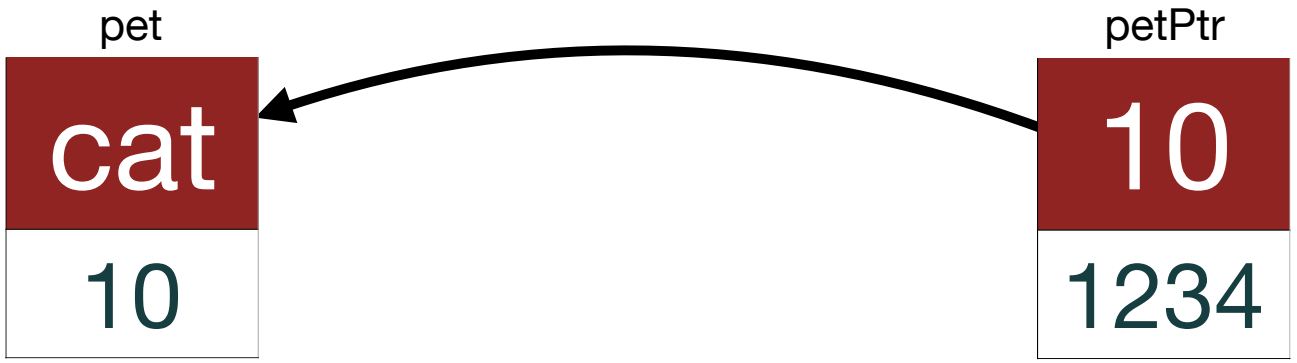
```
string *petPtr; // declare a pointer (which will hold a
                //                    memory address) to a string
string pet = "cat"; // a string variable

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:

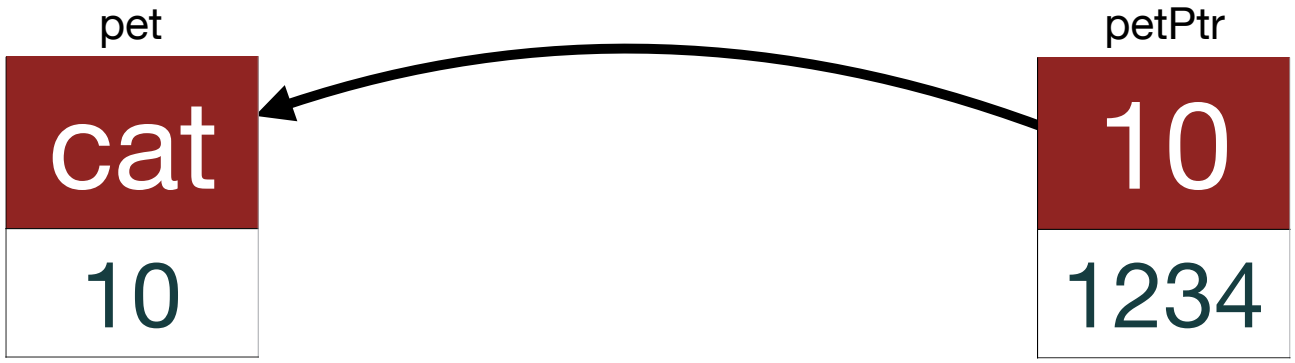
```
string *petPtr; // declare a pointer (which will hold a
                //                    memory address) to a string
string pet = "cat"; // a string variable

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



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"
```



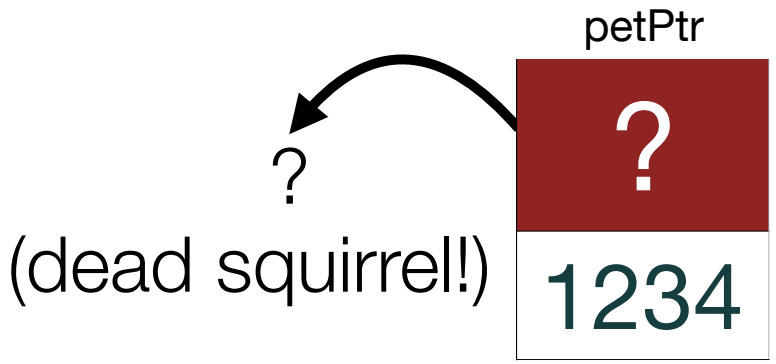
This is called "dereferencing" the pointer: the asterisk says, "go to where the pointer is pointing, and return the *value* stored there"

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
```

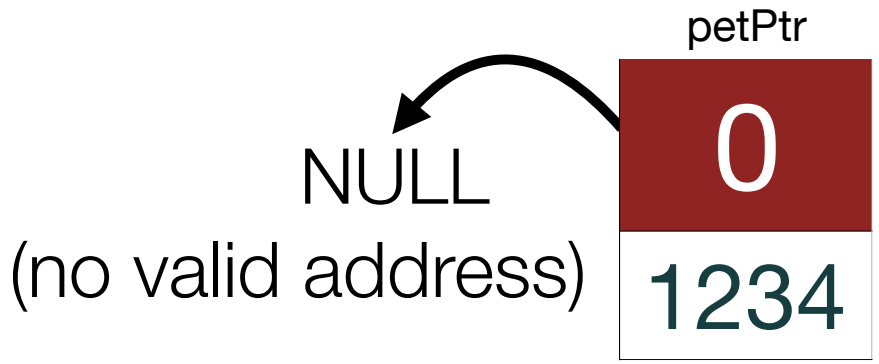


Pointer Tips

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;
```

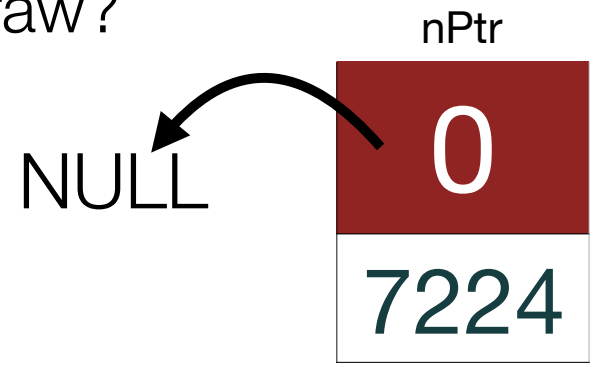
What type does this pointer point to? **an int**
What should we draw?

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?

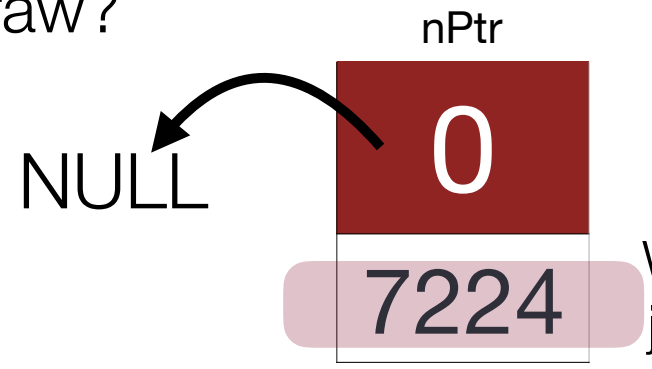


Pointer Practice

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?



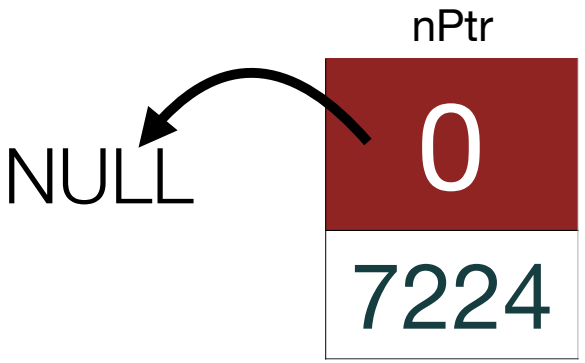
We don't care what this number is, just that it tells us where **`nPtr`** is in memory.

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;
```

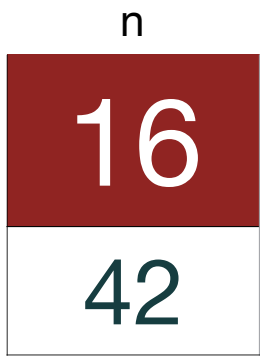
What should we draw?

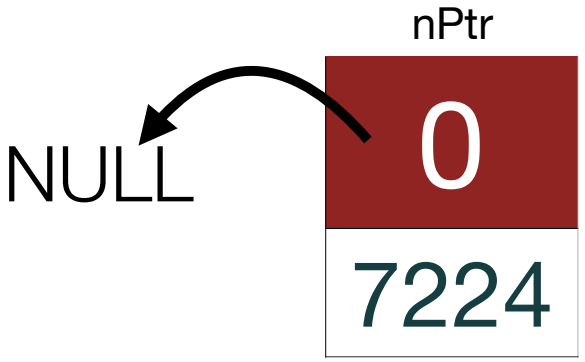


Pointer Practice

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;
```



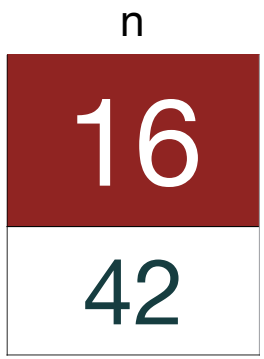


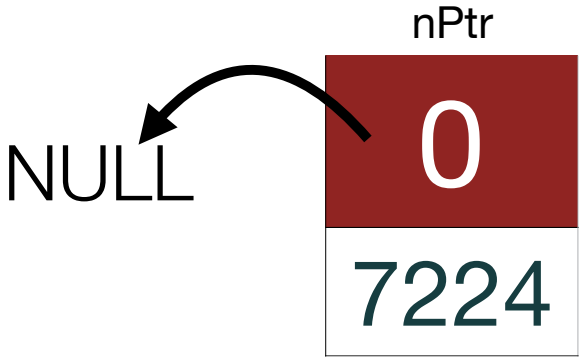
Pointer Practice

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?

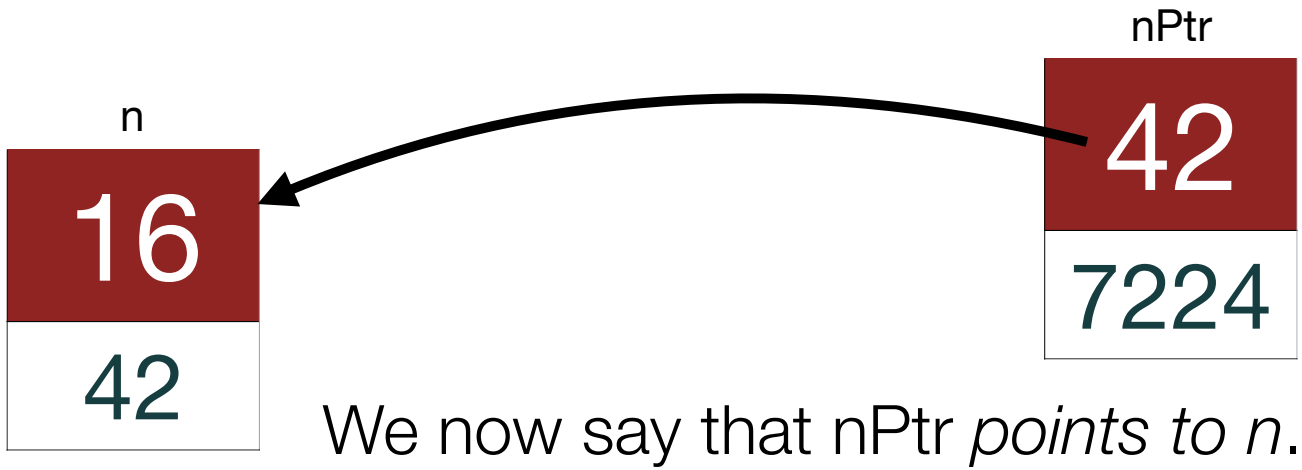




Pointer Practice

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*.

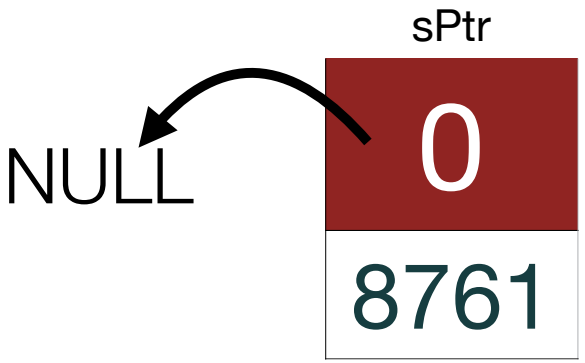
Pointers

What is a pointer??

a memory address!

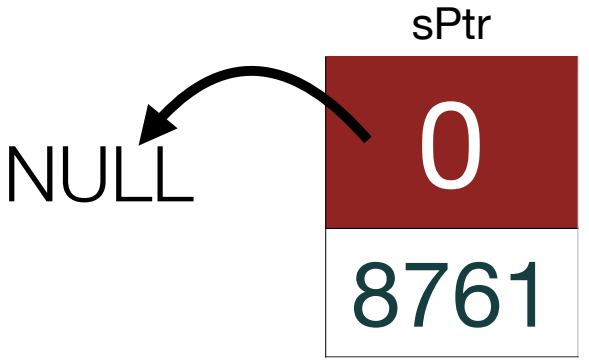
Pointer Practice

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



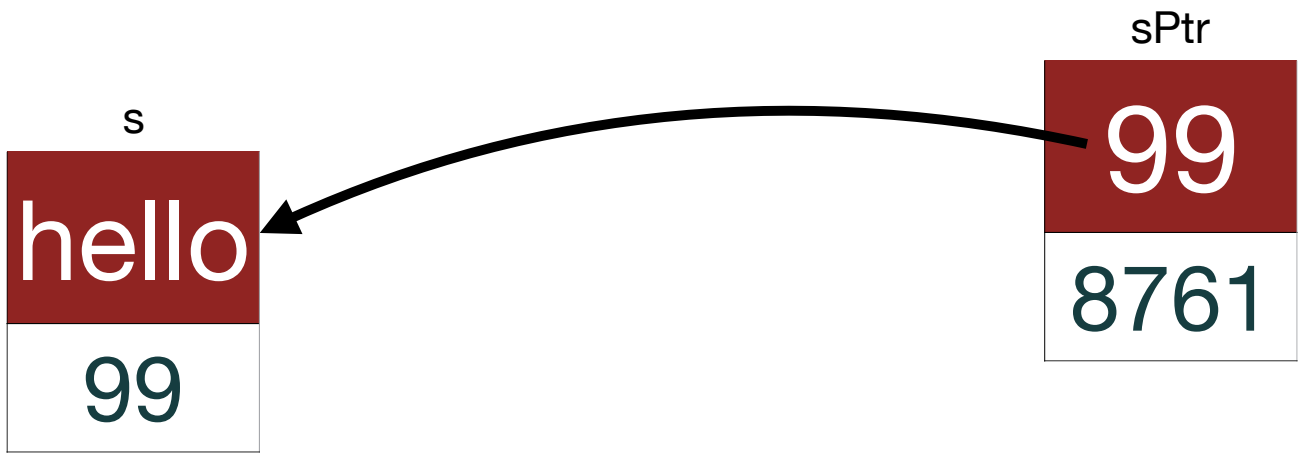
Pointer Practice

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



Pointer Practice

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

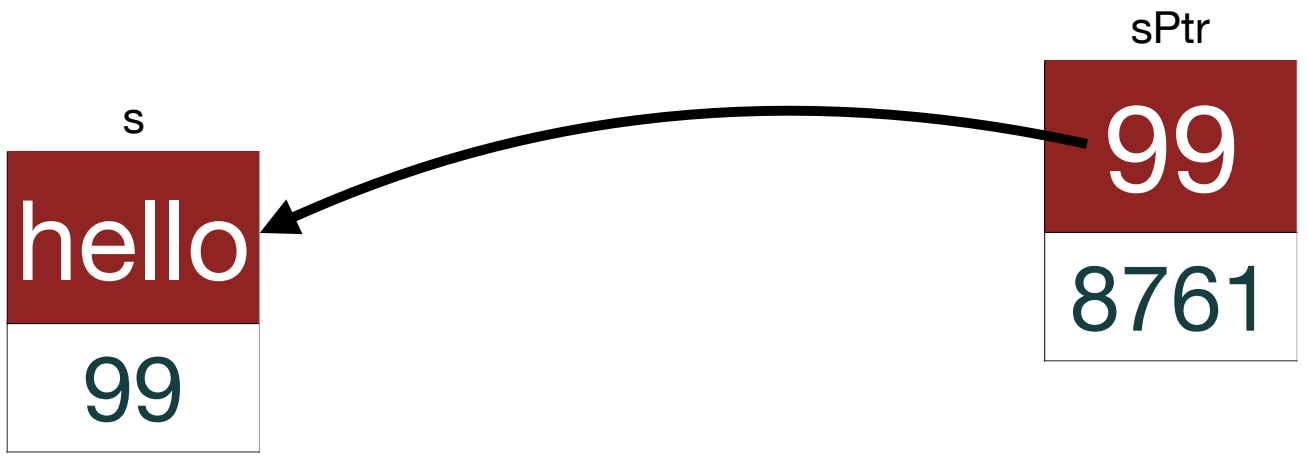


Pointer Practice

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

Output:

hello



Pointer Practice

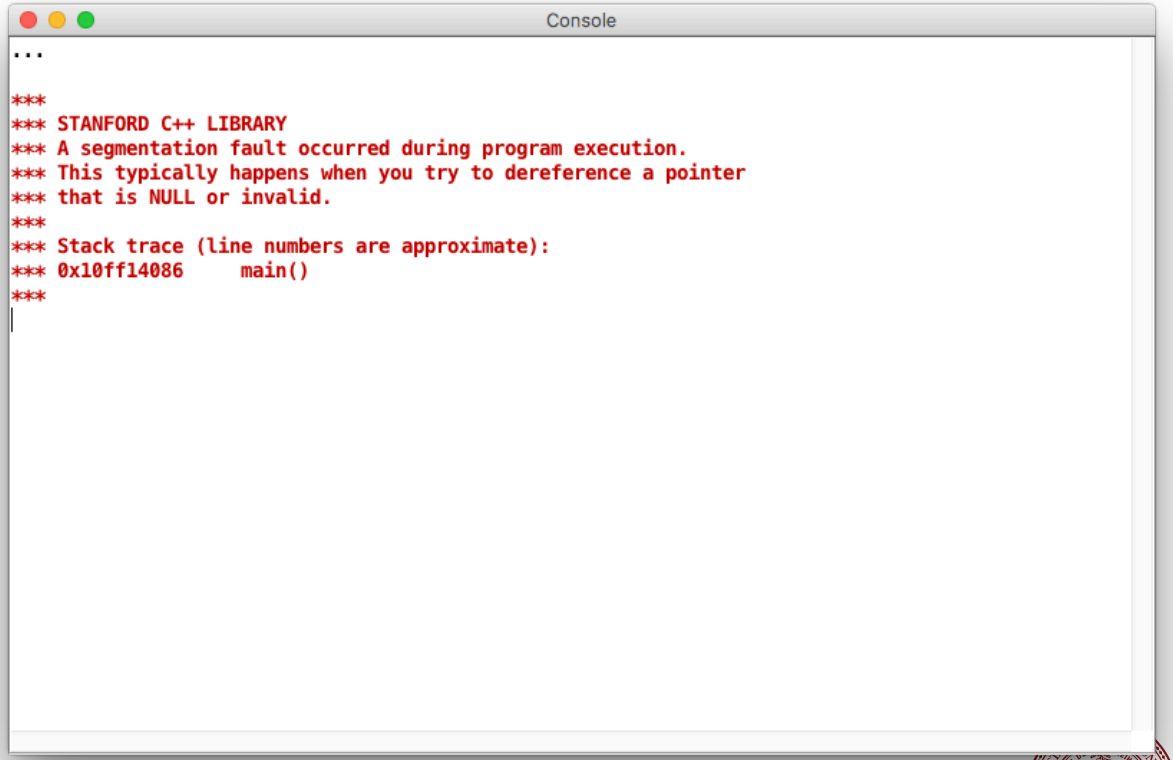
```
string *sPtr = NULL;
string s = "hello";
cout << *sPtr << endl;
```

Output?

Pointer Practice

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

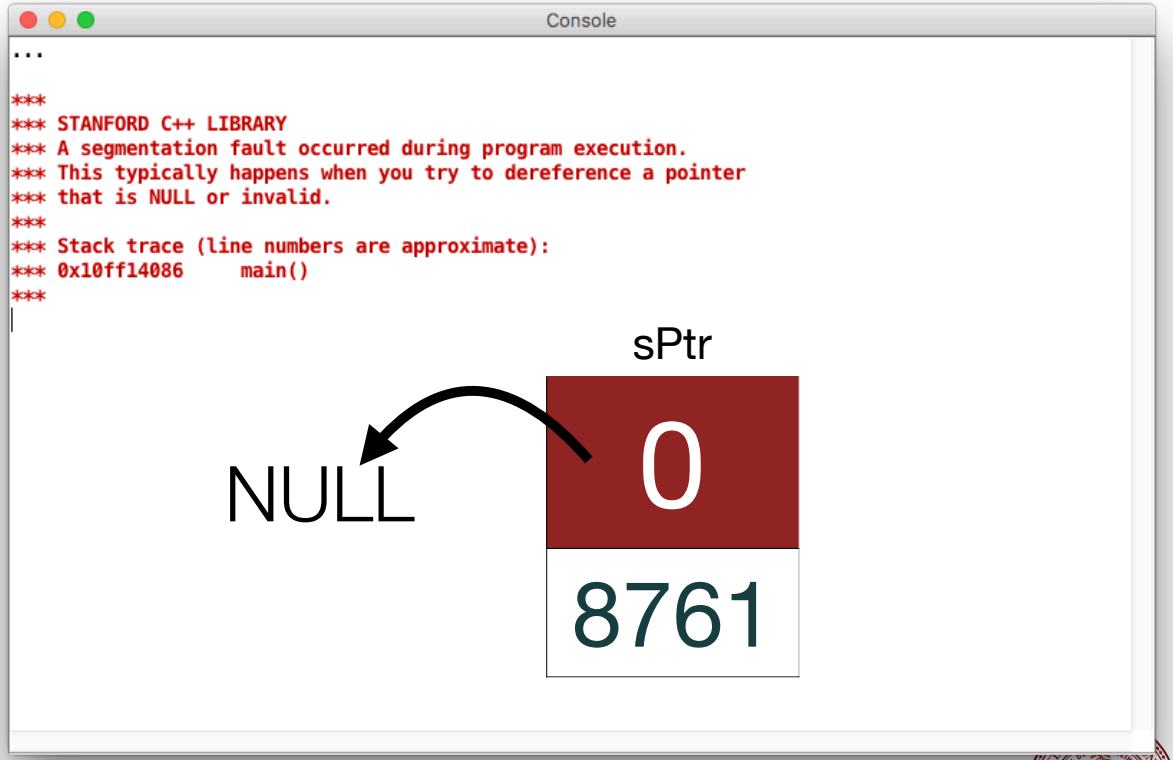
Output? Seg Fault! (crash!)



Pointer Practice

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

Output? Seg Fault! (crash!)



Be careful when dereferencing pointers!

Pointer Practice

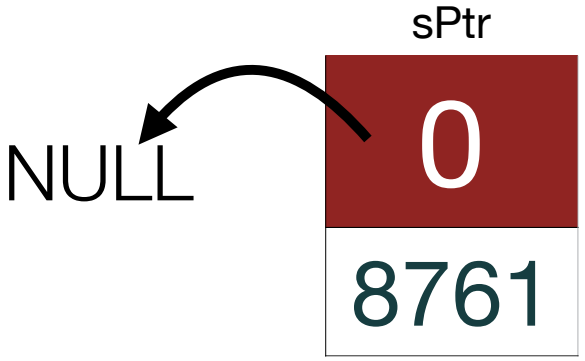
- You can also use the dereferencing operator to *set* the value of the "pointee" (the variable being pointed to):

```
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):

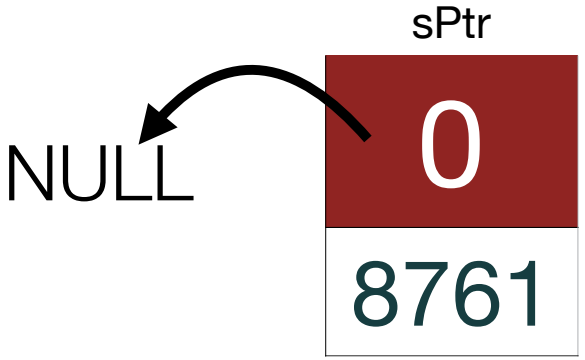
```
string *sPtr = NULL;
string s = "hello";
sPtr = &s;
*sPtr = "goodbye";
cout << s << endl;
```



Pointer Practice

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

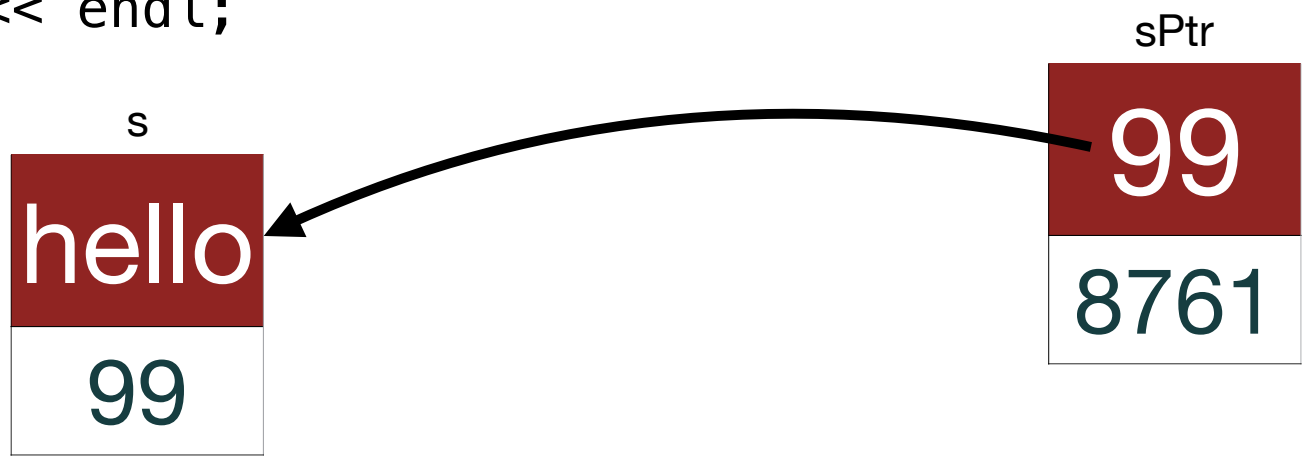
```
string *sPtr = NULL;
string s = "hello";
sPtr = &s;
*sPtr = "goodbye";
cout << s << endl;
```



Pointer Practice

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

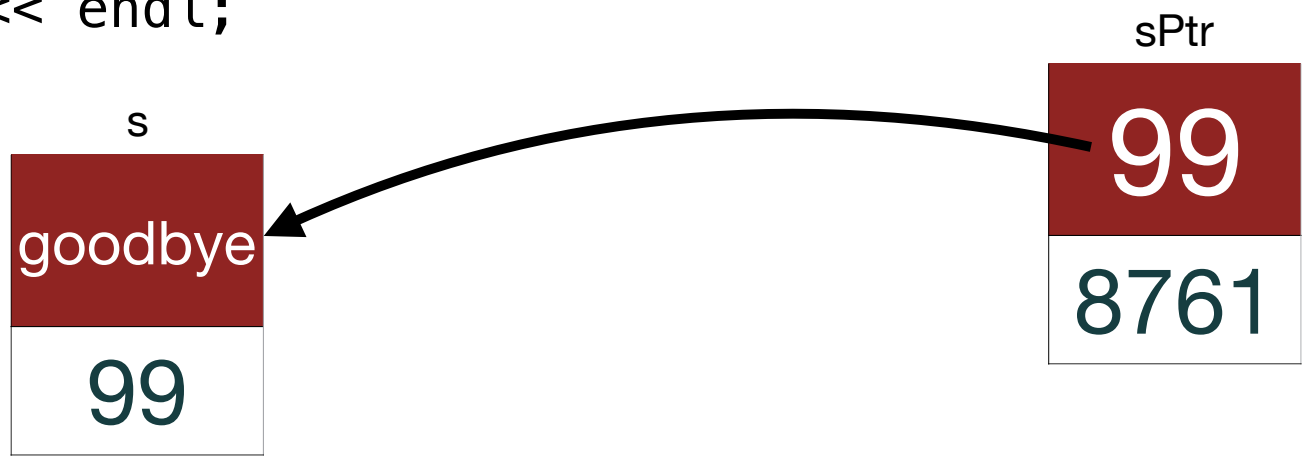
```
string *sPtr = NULL;
string s = "hello";
sPtr = &s;
*sPtr = "goodbye";
cout << s << endl;
```



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;
```



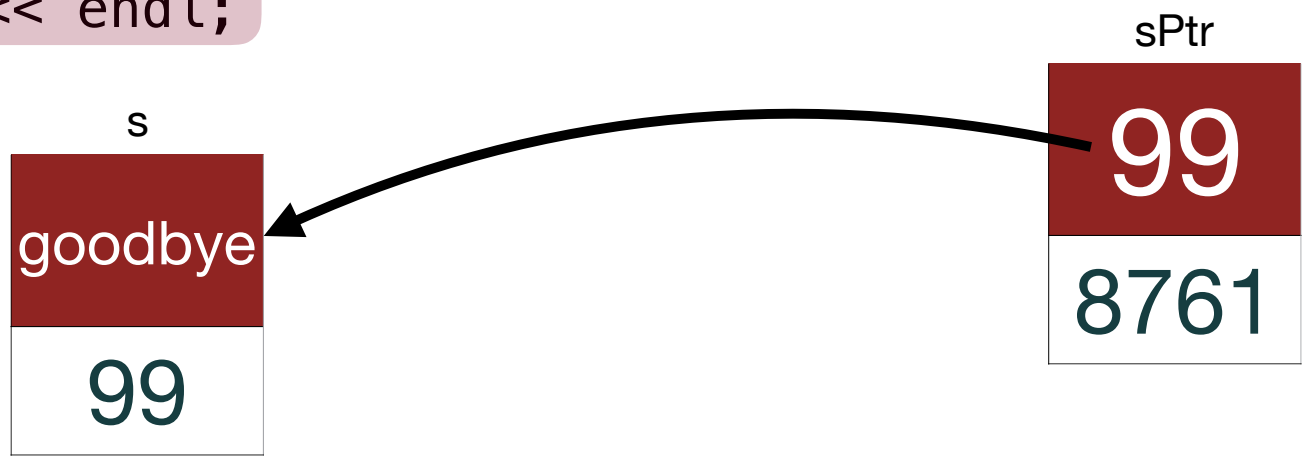
Pointer Practice

- 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:

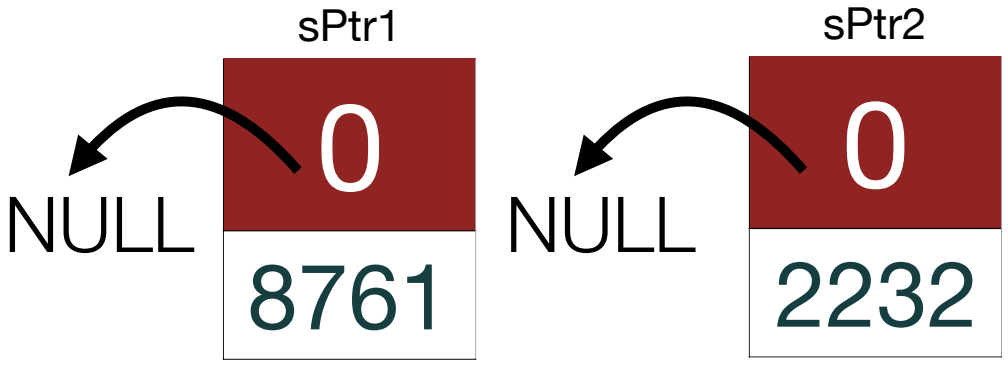
goodbye



Pointer Practice

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

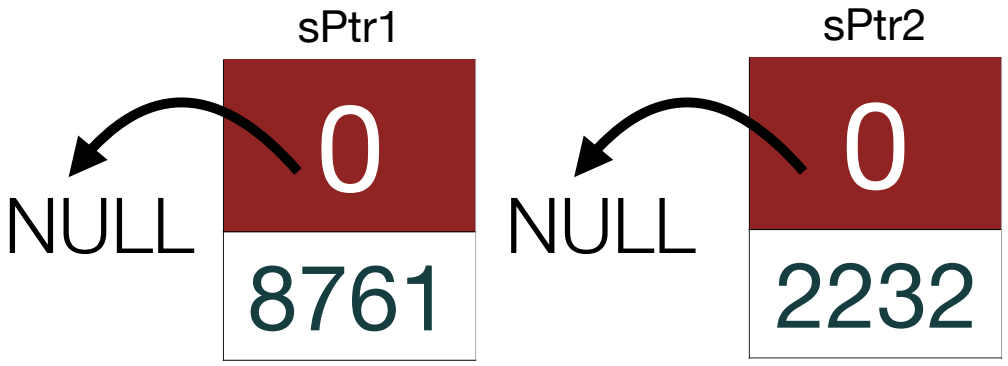
```
string *sPtr1 = NULL;
string *sPtr2 = NULL;
string s = "hello";
sPtr1 = &s;
cout << *sPtr1 << endl;

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



Pointer Practice

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

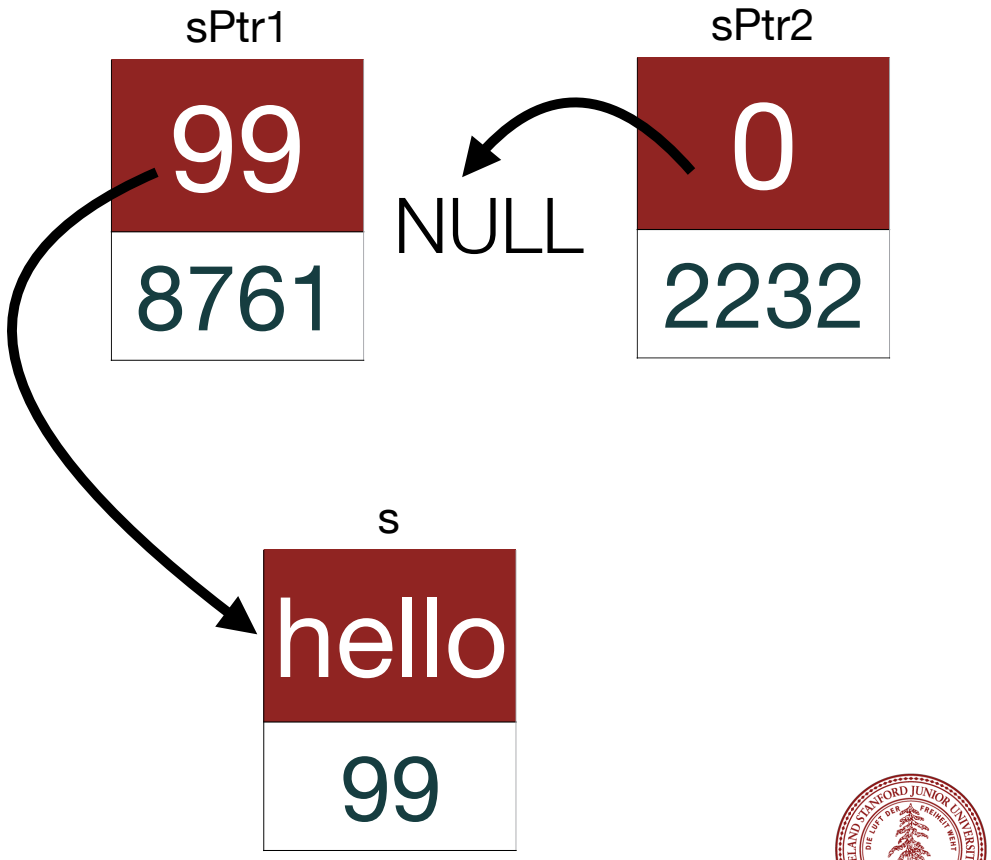
```
string *sPtr1 = NULL;
string *sPtr2 = NULL;
string s = "hello";
sPtr1 = &s;
cout << *sPtr1 << endl;

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;
```



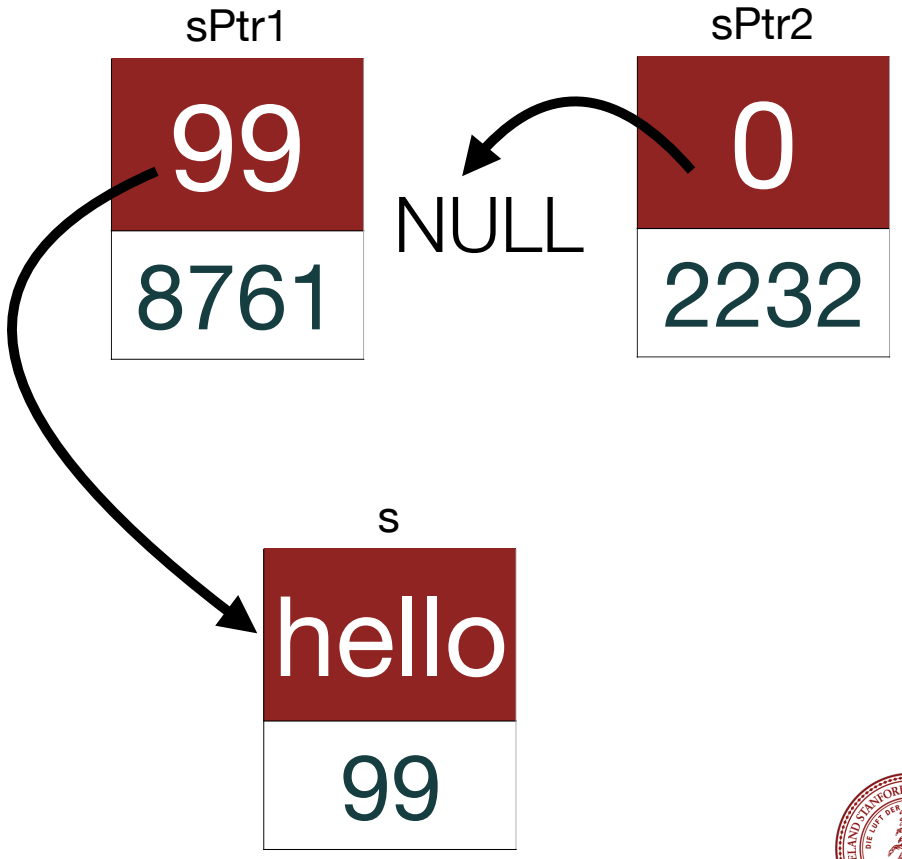
Pointer Practice

- 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:

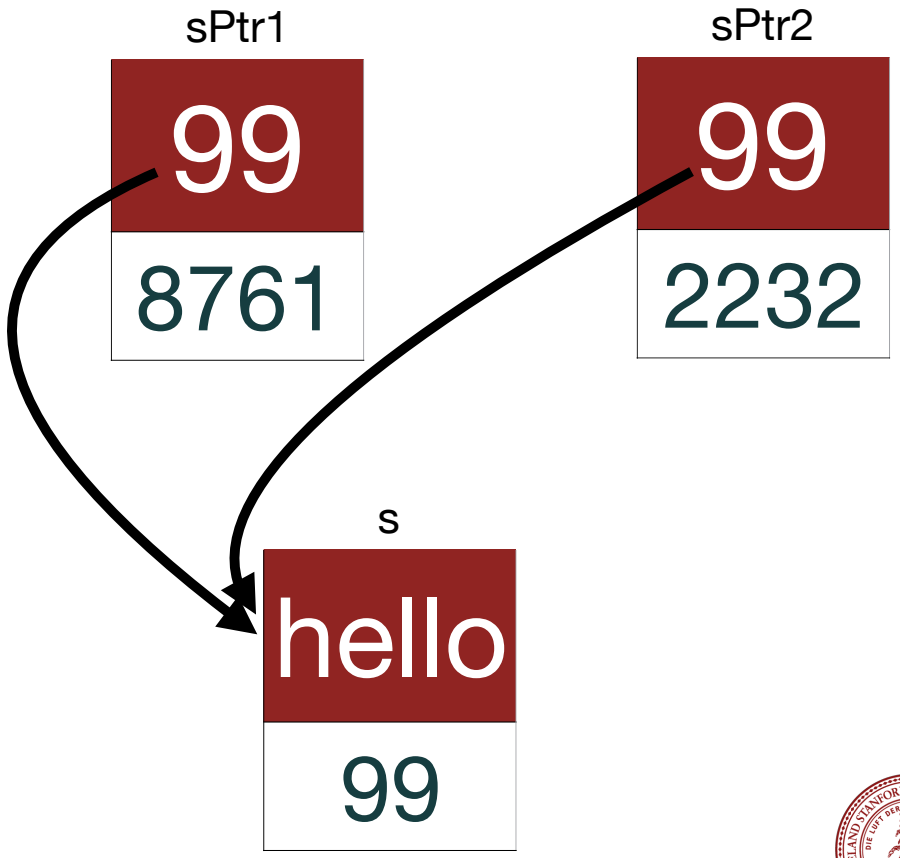
hello



Pointer Practice

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

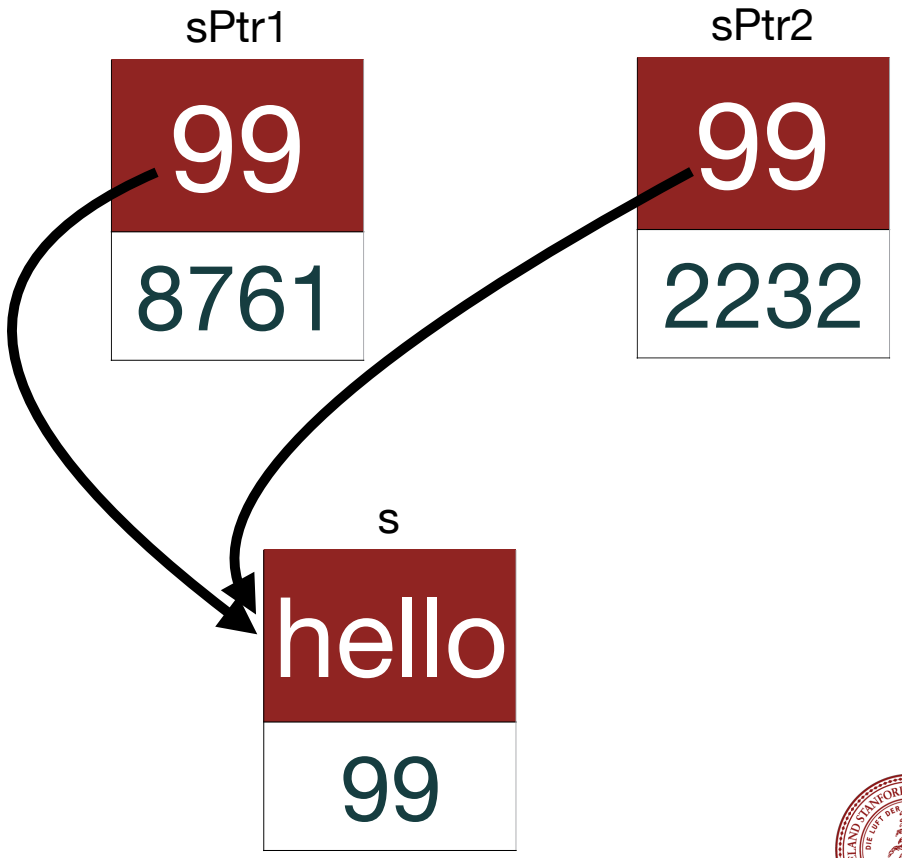
```
string *sPtr1 = NULL;
string *sPtr2 = NULL;
string s = "hello";
sPtr1 = &s;
cout << *sPtr1 << endl;

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;
```



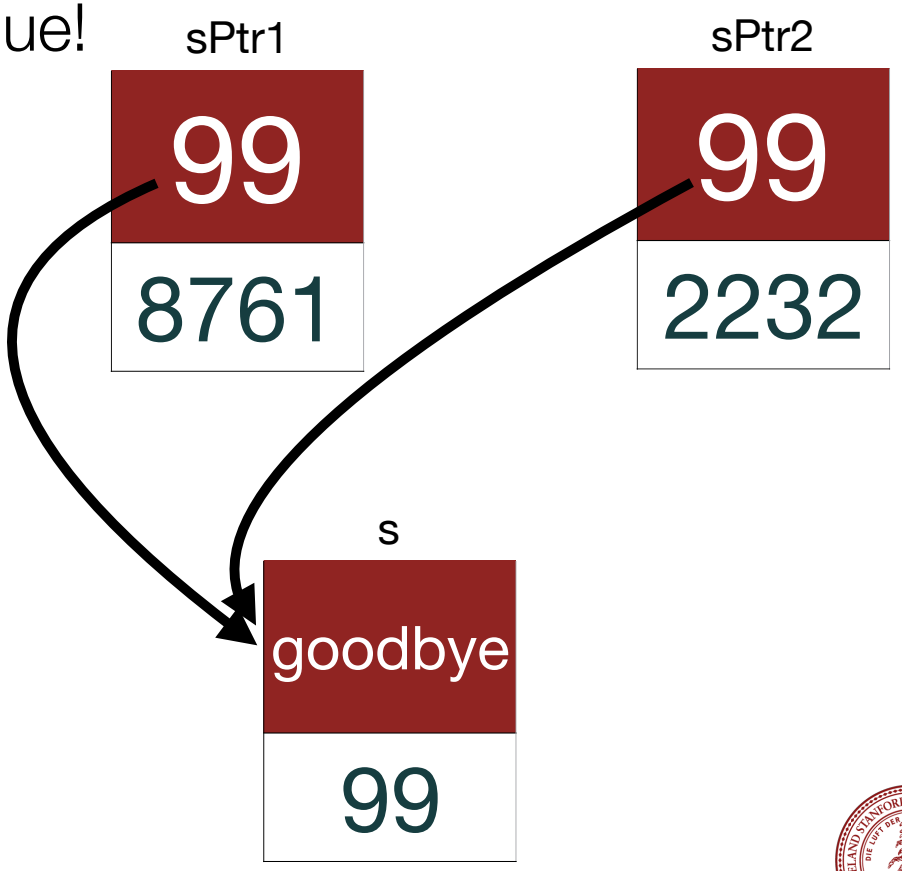
Output:

hello

Pointer Practice

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

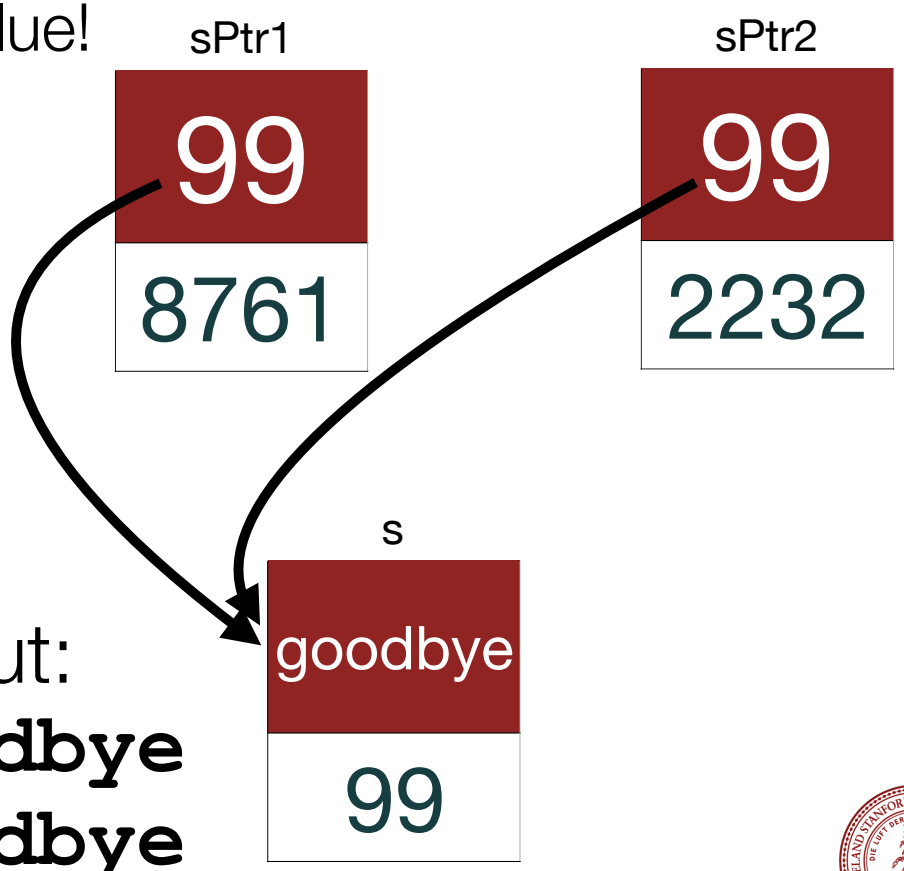
```
string *sPtr1 = NULL;
string *sPtr2 = NULL;
string s = "hello";
sPtr1 = &s;
cout << *sPtr1 << endl;

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`

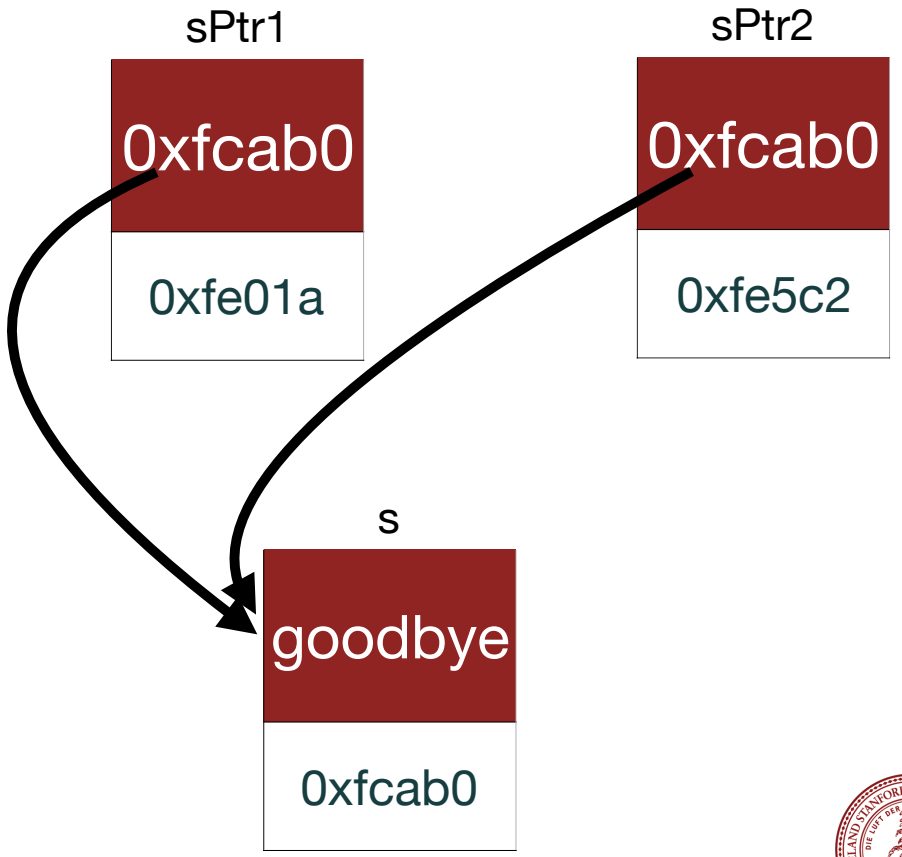
This is a base-16, or "hexadecimal" representation. The **`0x`** just means "the following number is in hexadecimal."

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.



Recap

- Classes:
  - public sections of your class are for everyone
  - private sections are for just your class
  - all class variables are shared among the class functions
  - the constructors are used when you create an *instance* of your class

- 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

Pointers and Structs!

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