

Pointers and Memory

An extremely powerful concept

With great power comes
great responsibility

With great power comes
great responsibility

- bad pointerism can lead to
 - extremely ugly crashes
 - difficult to debug glitches
 - random crashes

Have you used pointers before?

Why have pointers?

Have you used pointers before?

Why have pointers?

- Pointers allow different sections of code to share information easily.
- Pointers enable complex 'linked' data structures

Simple variables

iCount



42

An int variable is like a box
which can store a single int
value

Pointers

- Don't store value directly
- Instead, stores a reference to another value (the pointee)

pointers

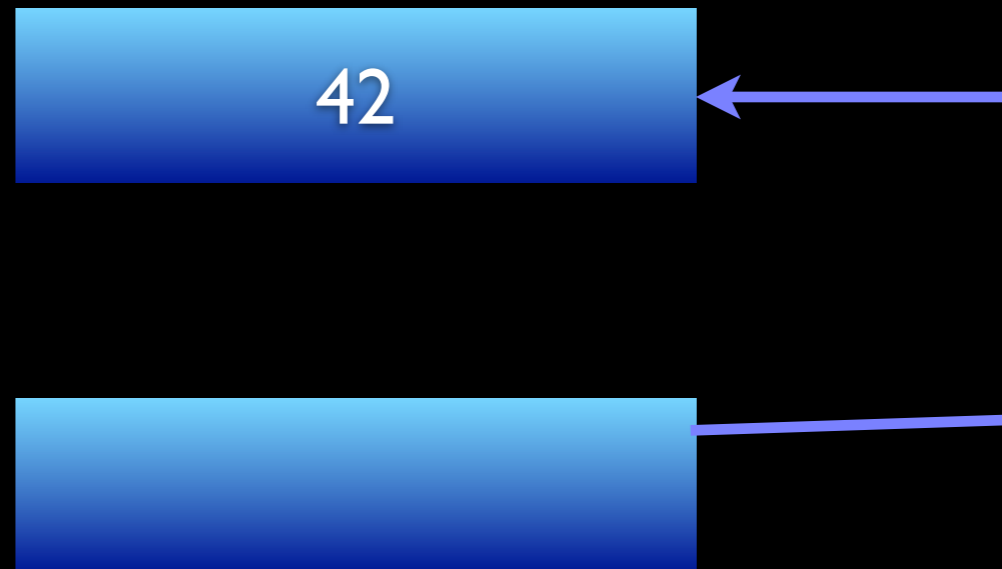
A simple int variable. Current val. 42

iCount

42

piCurrent

A pointer variable.
Current value is a reference to iCount



The magic wand of dereferencing

How do I follow a pointer's reference to get the value of the pointee?



```
#include<iostream>

using namespace std;

int main(){
    int iCount = 42;
    int* piCurrent = &iCount;

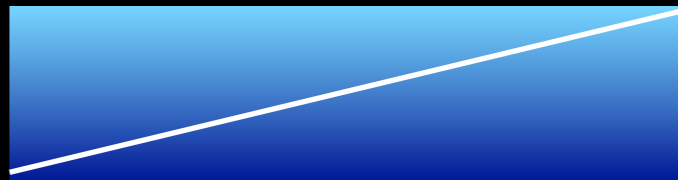
    cout << *piCurrent << endl;

}
```

The null potion

How do I point to nothing?

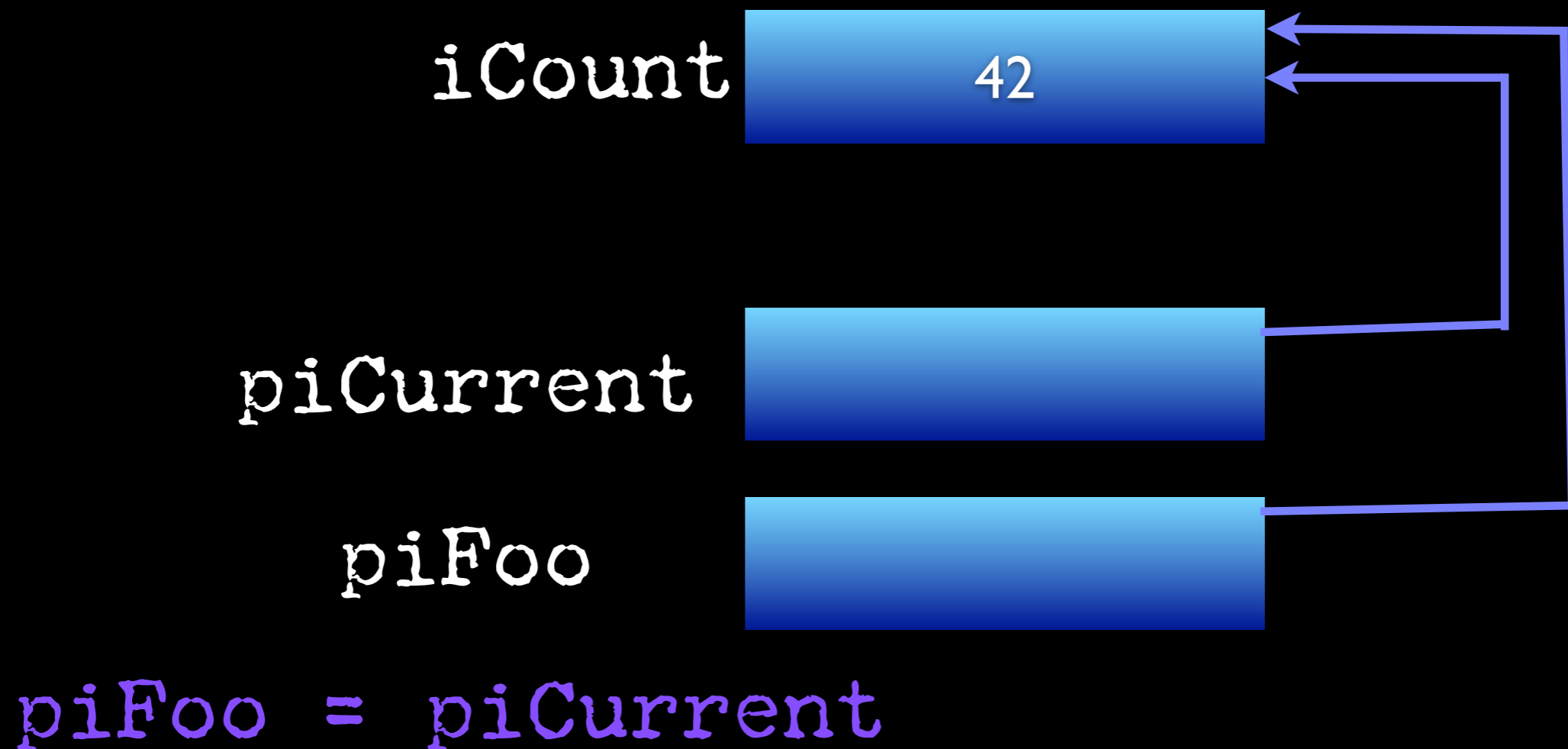
piCurrent



C: NULL (constant 0)
C++: 0
Java: null

The magic wand of pointer assignment

- The assignment operator (=) between 2 pointers makes them point to the same pointee.



SHARING:

two pointers referring to a single
pointee are said to be sharing.

2 entities share a single memory
structure

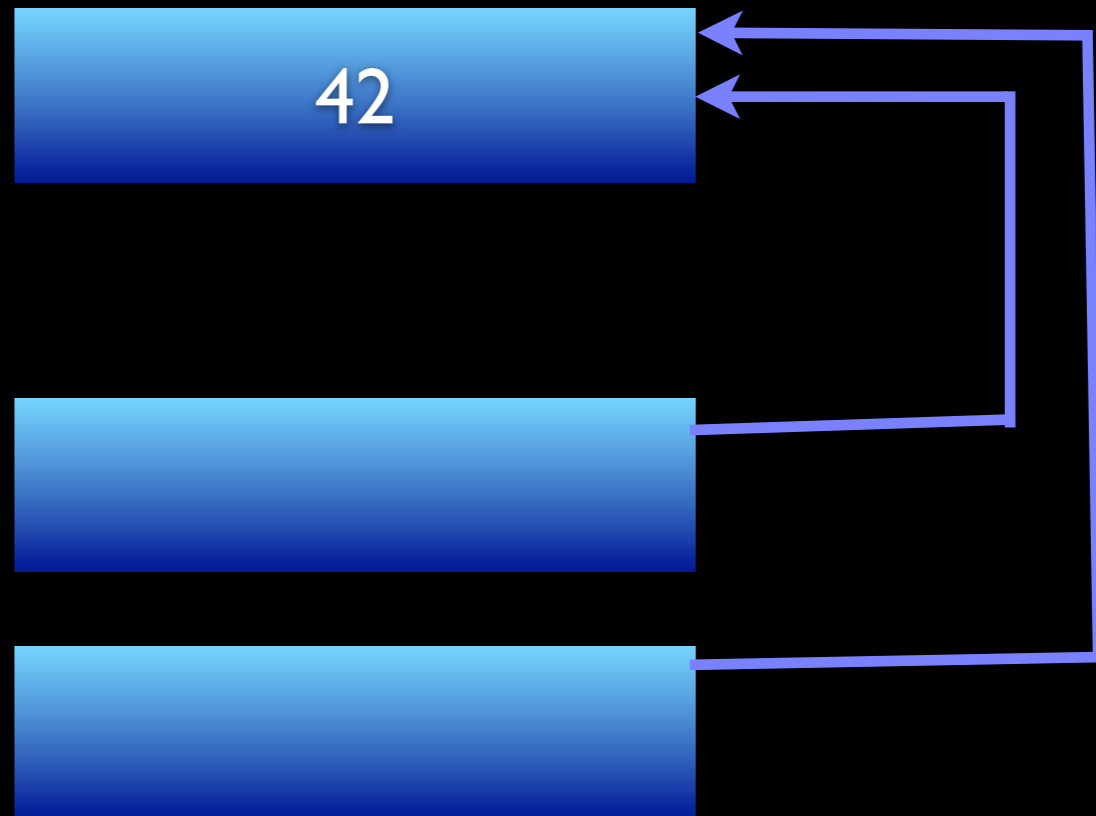
iCount

42

piCurrent

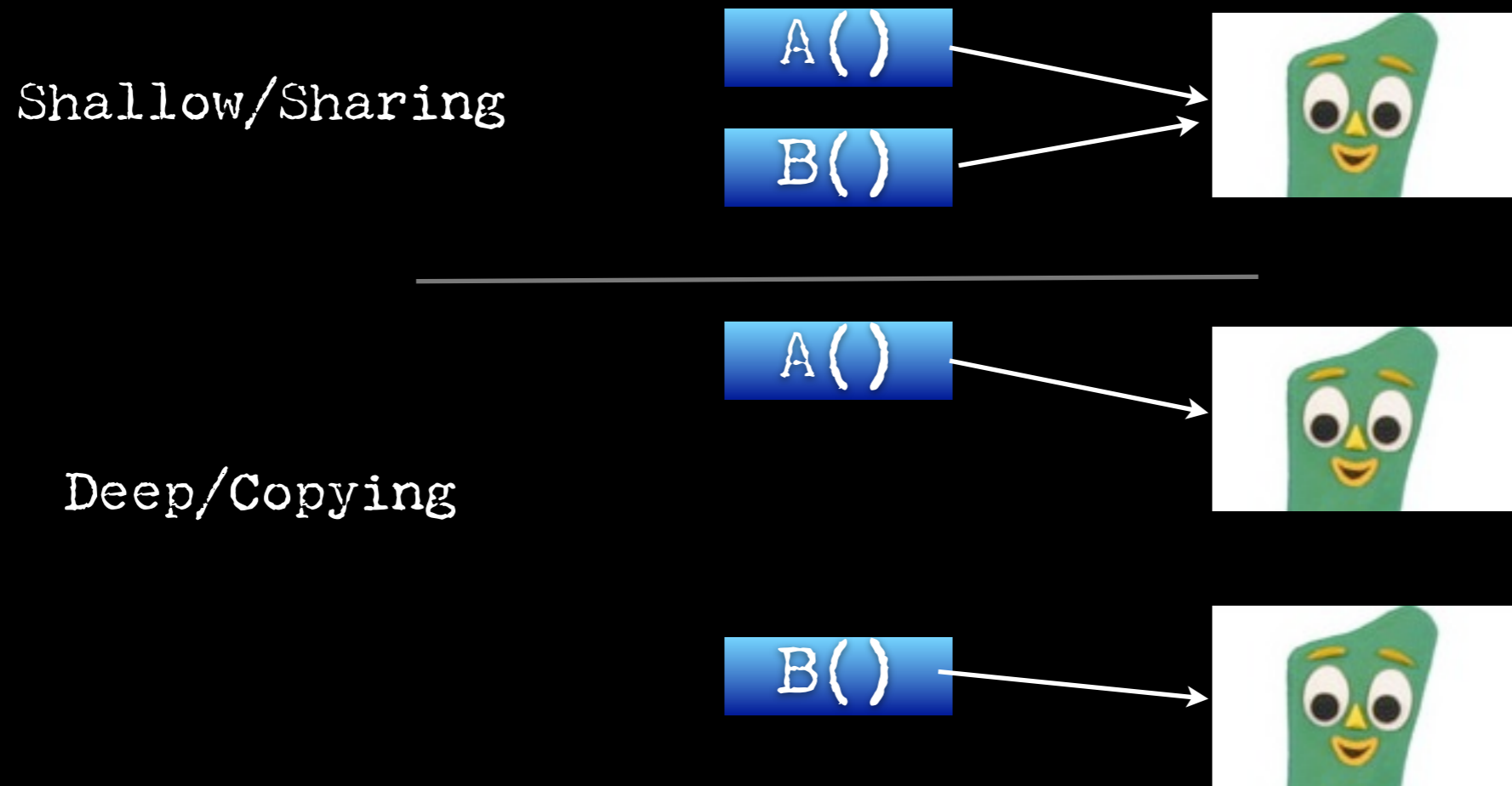
piFoo

piFoo = piCurrent



Shallow and Deep Copying

- How do functions pass values?



Good News

In C++ pointers are inherently good.

Bad pointers

- In C every pointer starts off bad.
- Must use magic to make it “good”

piFoo

@*&*%\$

- Bad pointer errors are very common.

Bad pointers

- In C pointers are inherently bad.
- In C++, Java, Perl, and Lisp pointers are inherently good (set to null)

That's everything conceptually you need to know about pointers

That's everything conceptually you need to know about pointers

1. the pointer must be allocated

That's everything conceptually you need to know about pointers

1. the pointer must be allocated
2. the pointee must be allocated

That's everything conceptually you need to know about pointers

1. the pointer must be allocated
2. the pointee must be allocated
3. the pointer must be assigned to point to the pointee

That's everything conceptually you need to know about pointers

1. the pointer must be allocated
2. the pointee must be allocated
3. the pointer must be assigned to point to the pointee
4. people rarely screw up (1)

That's everything conceptually you need to know about pointers

<syntax>

Syntax

- A pointer type in C is just the pointee type followed by *

```
int *  
float *  
char *
```

Syntax

- A pointer type in C is just the pointee type followed by *

```
int *  
float *  
char *
```

Variables

```
int * piFoo;
```

allocates space for the pointer
but not the pointee
pointer starts out NULL

Try this code . . .

```
#include<iostream>

using namespace std;

int main(){
    int* piCurrent;
    *piCurrent = 42;

    cout << *piCurrent << endl;
}
```

compile and run

Bus Error 10
memory protection error

The & operator (reference to)

```
void numPtrExample() {  
    int iCount;  
    int* piFoo;  
  
    iCount = 42  
    piFoo = &iCount;  
}
```

The & computes a reference to the variable

The & operator (reference to)

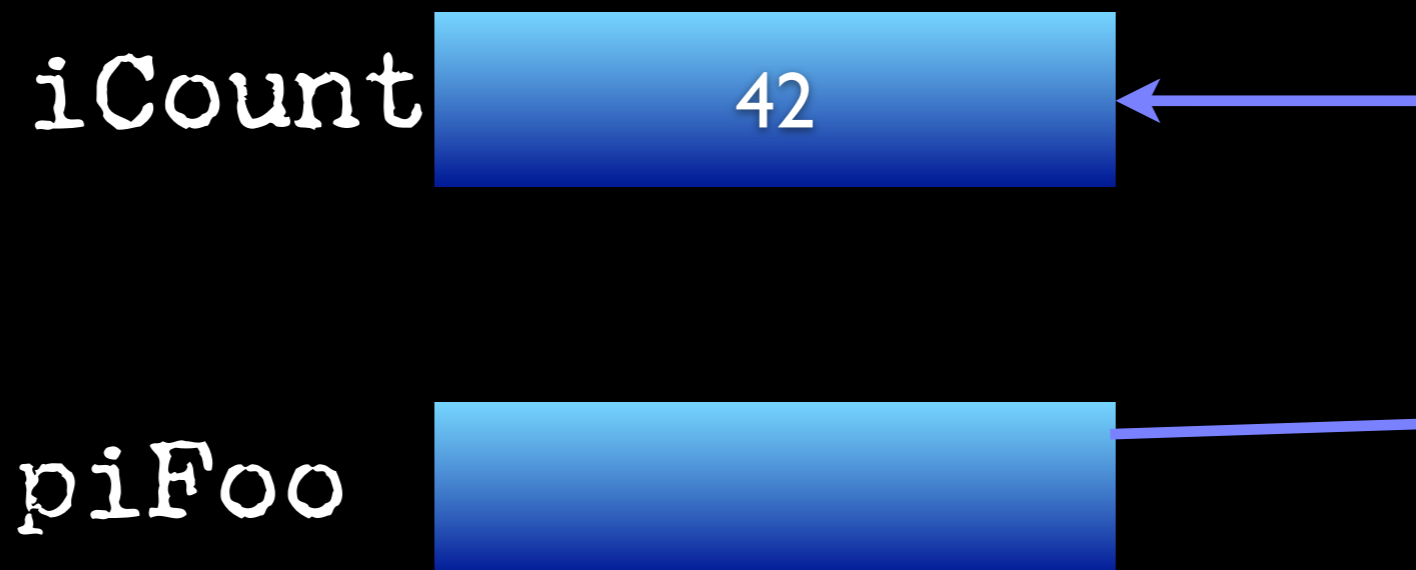
```
void numPtrExample() {  
    int iCount;  
    int* piFoo;  
  
    iCount = 42  
    piFoo = &iCount;  
}
```



The & operator (reference to)

It is possible to use &
in a way that compiles
fine but creates a
problem @ run time....

more on this later

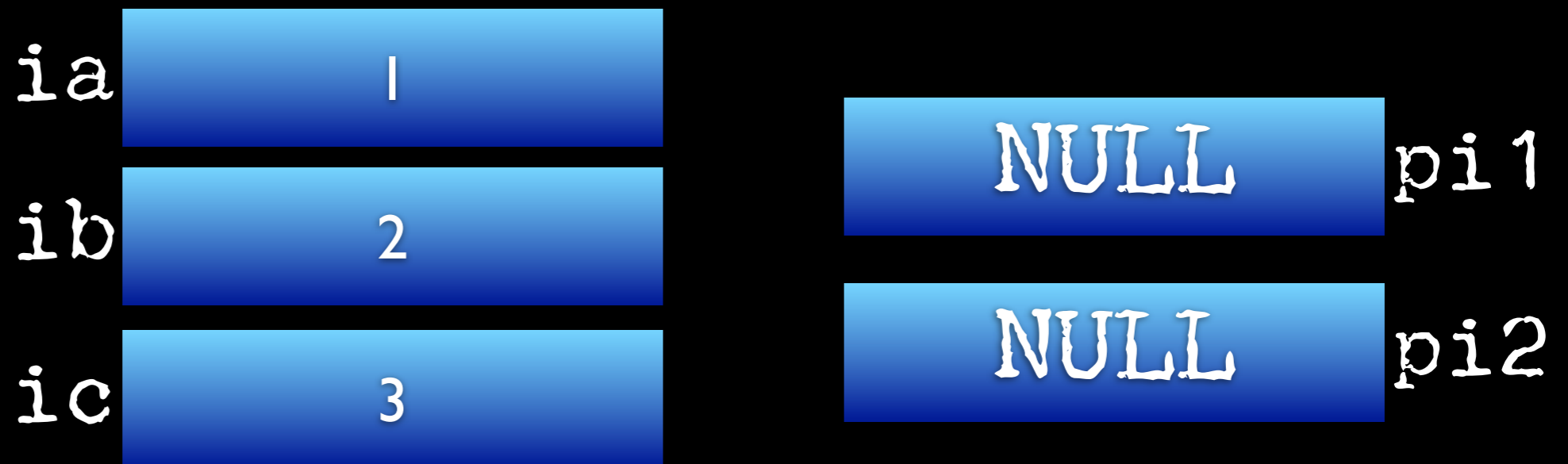


The * operator dereferences

```
void pointerTest() {  
    int ia = 1;  
    int ib = 2;  
    int ic = 3;  
    int* pi1;  
    int* pi2;
```

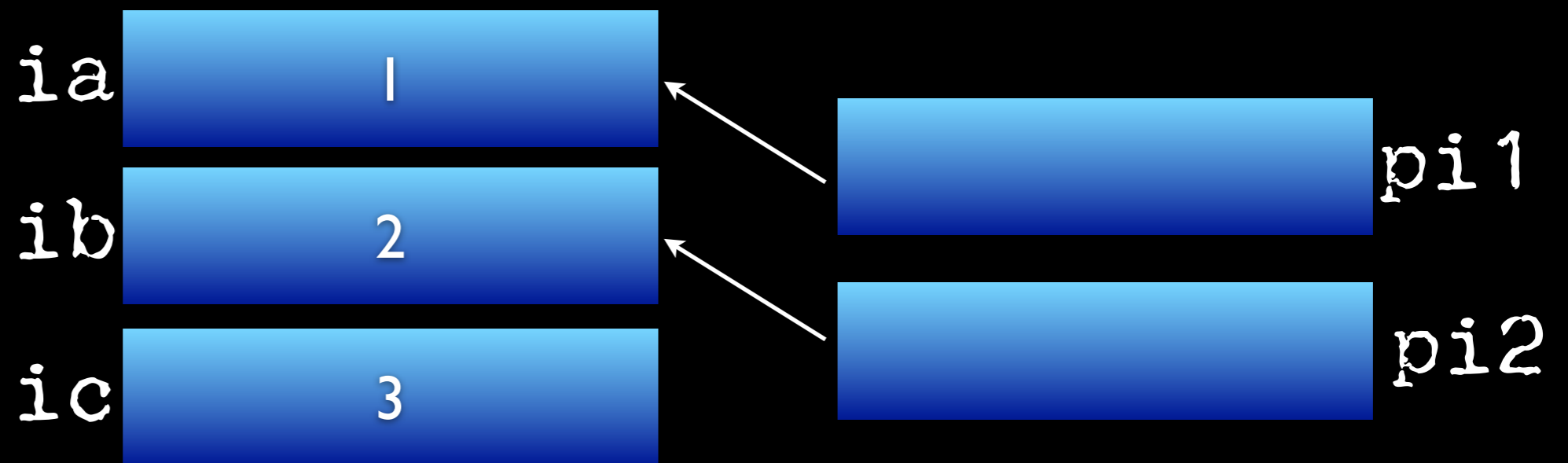
The * operator dereferences

```
void pointerTest() {  
    int ia = 1;  
    int ib = 2;  
    int ic = 3;  
    int* pi1;  
    int* pi2;  
}
```



The & operator provides a reference to

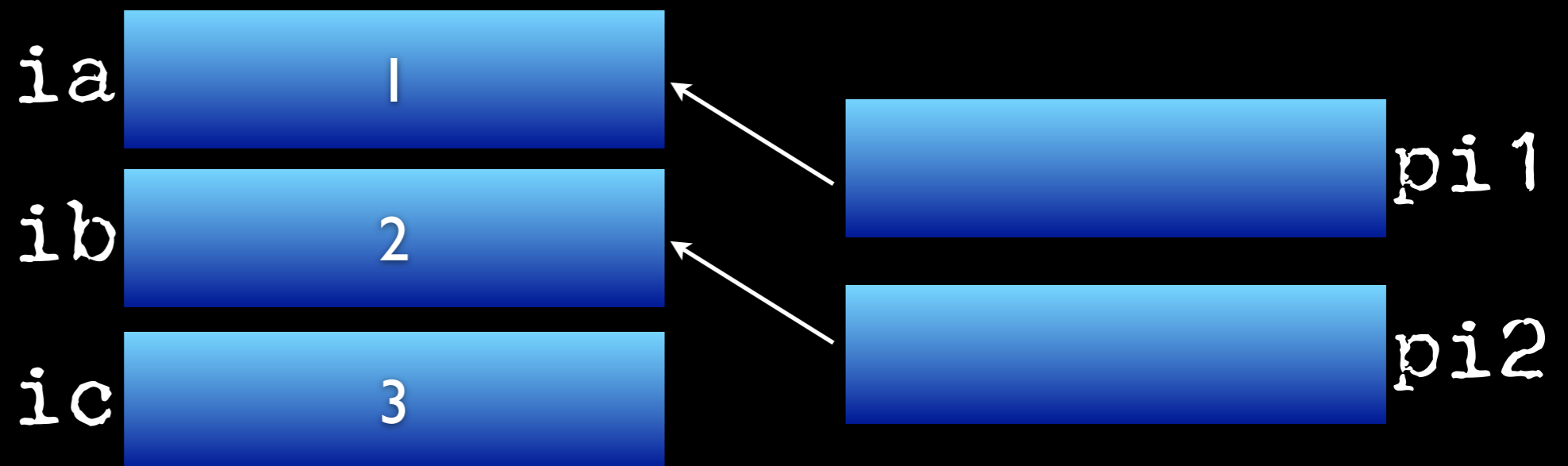
```
pi1 = &ia;  
pi2 = &ib;
```



The * operator dereferences

```
pi1 = &ia;  
pi2 = &ib;
```

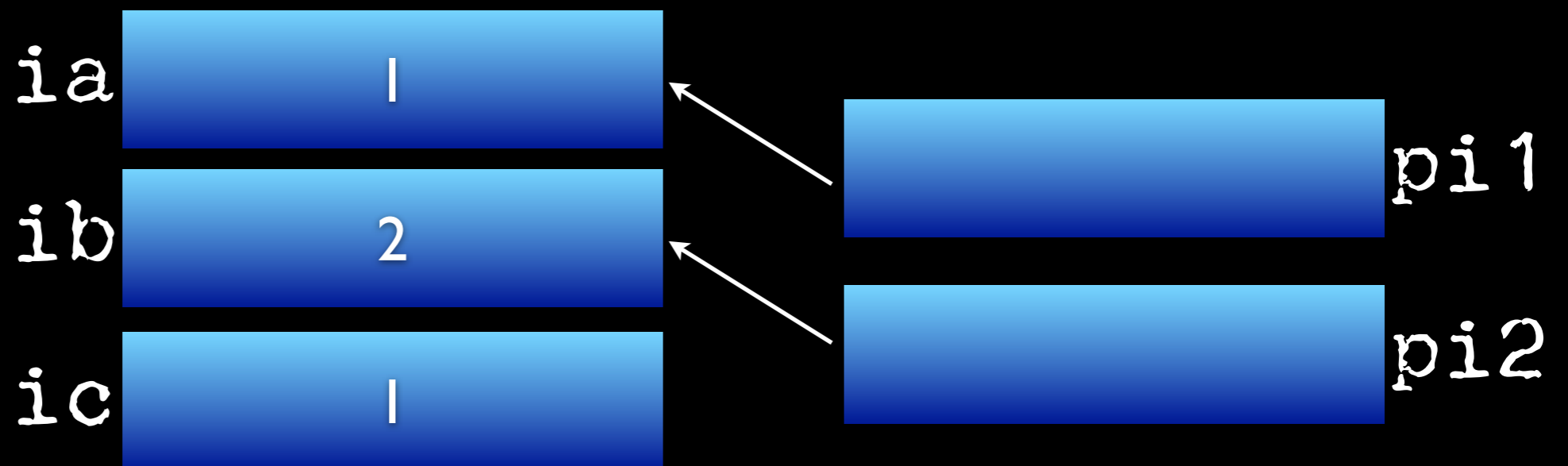
```
ic = *pi1; /* what does this do?*/
```



The * operator dereferences

```
pi1 = &ia;  
pi2 = &ib;
```

```
ic = *pi1; /* what does this do?*/
```

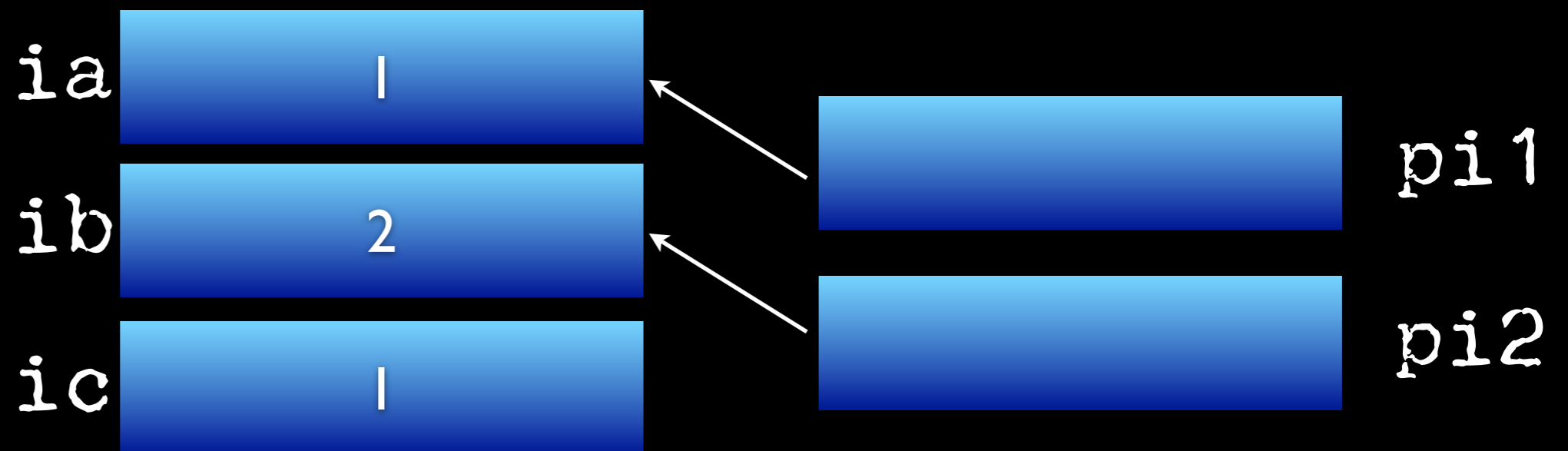


The * operator dereferences

```
pi1 = &ia;  
pi2 = &ib;
```

```
ic = *pi1;  
pi1 = pi2;  
*pi1 = 13;
```

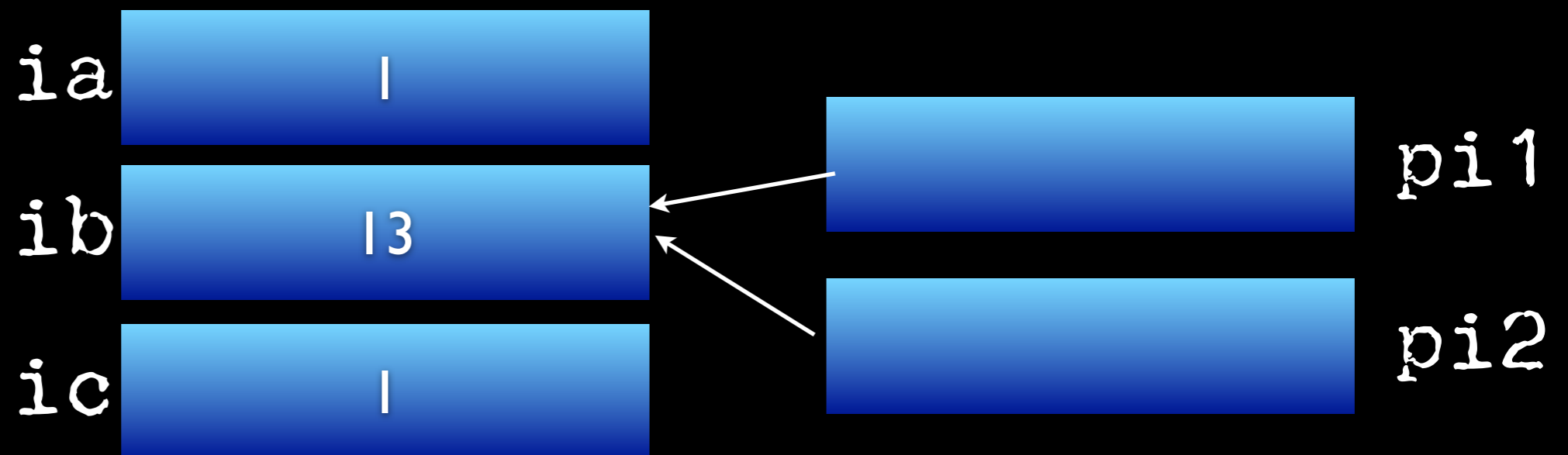
```
/* ????? */
```



The * operator dereferences

```
pi1 = &ia;  
pi2 = &ib;
```

```
ic = *pi1;  
pi1 = pi2;  
*pi1 = 13;
```



Another example

```
void pointerEx2() {  
    int* piFoo;  
    *piFoo = 42;  
}
```


Another example

```
void pointerEx2() {  
    int* piFoo;  
    *piFoo = 42;  
}
```

piFoo

@*&*%\$



Summary

- A pointer stores a reference to its pointee. The pointee, in turn, stores something useful.

Summary

- The dereference operation on a pointer accesses its pointee.
- A pointer may only be dereferenced after it has been assigned to refer to a pointee.
- Most pointer bugs involve violating this rule.

Summary

- Allocating a pointer does not automatically assign it to refer to a pointee.
- Assigning the pointer to refer to a specific pointee is a separate operation which is easy to forget.

Summary

- Assignment between two pointers makes them refer to the same pointee which introduces sharing.

<Local Memory>

Everyone uses local variables
but no one thinks about them.

You have a bunch of local
variables in your code

```
int ilength;  
int isum;  
float fFrequency;  
int itotal;  
int iItemsSold;
```

All these represent space in
computer memory

You have a bunch of local variables in your code

```
int ilength;  
int isum;  
float fFrequency;  
int itotal;  
int iItemsSold;
```

All these represent space in computer memory

It's not the case that every variable in a program has a permanently assigned area of memory

You have a bunch of local
variables in your code

```
int ilength;  
int isum;  
float fFrequency;  
int itotal;  
int iItemsSold;
```

Modern compilers are smart enough to
give memory to a variable only when
necessary

The terminology is allocate and
deallocate

The most common memory related error is using a deallocated variable.

Local Memory

```
int Square(int inum) {  
    int ireresult;  
    ireresult = inum * inum;  
    return ireresult;  
}
```

When the square function runs, memory is allocated for inum and ireresult.

When the function exits, the storage is deallocated.

```
void Foo(int ifoo) {
    int i;
    float fScores[100];
    ifoo = ifoo + 1;
    for (i = 0; i < ifoo; i++)
    {
        Bar(i + ifoo);
    }
}
```

ifoo, i, fScores, allocated when Foo runs.

These variables continue to exist within the for loop.

They continue to exist even during calls to other fns.

The locals are deallocated when the fn. exits

a slightly more complex example

First:

Each running program on your computer has its own:

memory space for the code
memory space for the stack
memory space for the heap

```
void X() {
    int ifoo = 1;
    int ibar = 2;
    // T1

    Y(ifoo);
    // T3

    Y(ibar);
    // T5
}

void Y(int iarg)
{
    int isum;
    isum = iarg + 2;
    // T2 (1st time) T4 (2nd)
}
```



```
void X() {  
    int ifoo = 1;  
    int ibar = 2;  
    // T1  
  
    Y(ifoo);  
    // T3  
  
    Y(ibar);  
    // T5  
}
```

```
void Y(int iarg)  
{  
    int isum;  
    isum = iarg + 2;  
    // T2 (1st time) T4 (2nd)  
}
```

T1

X()

ifoo

1

ibar

2

```
void X() {
    int ifoo = 1;
    int ibar = 2;
    // T1

    Y(ifoo);
    // T3

    Y(ibar);
    // T5
}
```

```
void Y(int iarg)
{
    int isum;
    isum = iarg + 2;
    // T2 (1st time) T4 (2nd)
}
```

T2

Y()

iarg

1

isum

3

X()

ifoo

1

ibar

2

```
void X() {
    int ifoo = 1;
    int ibar = 2;
    // T1

    Y(ifoo);
    // T3

    Y(ibar);
    // T5
}
```

```
void Y(int iarg)
{
    int isum;
    isum = iarg + 2;
    // T2 (1st time) T4 (2nd)
}
```

T3

X()

ifoo

1

ibar

2

```

void X() {
    int ifoo = 1;
    int ibar = 2;
    // T1

    Y(ifoo);
    // T3

    Y(ibar);
    // T5
}

```

```

void Y(int iarg)
{
    int isum;
    isum = iarg + 2;
    // T2 (1st time) T4 (2nd)
}

```

T4

Y()

iarg

2

isum

4

X()

ifoo

1

ibar

2

```
void X() {  
    int ifoo = 1;  
    int ibar = 2;  
    // T1  
  
    Y(ifoo);  
    // T3  
  
    Y(ibar);  
    // T5  
}
```

```
void Y(int iarg)  
{  
    int isum;  
    isum = iarg + 2;  
    // T2 (1st time) T4 (2nd)  
}
```

T5

X()

ifoo

1

ibar

2

Advantages of Locals

- Convenient- fns. usually need temp. memory.
- efficient - allocating and deallocating fast
- local copies - maintains independence

Disadvantages of Locals

- Short lifetime - sometimes want things to last beyond life of fn.
- Restricted Communication (flipside of independence)

Example

```
// Returns pointer to int
int * piMakeNode() {
    int itemp = 0;
    return (&itemp);
}

void Victim() {
    int * piFoo;
    piFoo = piMakeNode();
    *piFoo = 42;
}
```


Example

```
// Returns pointer to int
int * piMakeNode() {
    int itemp = 0;
    return (&itemp);
}

void Victim() {
    int * piFoo;
    piFoo = piMakeNode();
    *piFoo = 42;
}
```

Is there a problem?

Comment on This Example

```
int ifactorial(int i) {  
    int inum;  
    inum = i * ifactorial(i - 1);  
    return inum;  
}
```

```
int main() {  
    int inumber;  
    inumber = ifactorial(5);  
    printf("num %d\n", inumber);  
}
```

Predictions?

Example

```
int ifactorial(int i) {
    int inum;
    inum = i * ifactorial(i - 1);
    return inum;
}

int main() {
    int inumber;
    inumber = ifactorial(5);
    printf("num %d\n", inumber);
}
```

Stack Overflow Error
Segmentation Fault

<reference parameters>

Caller-Callee Communication

- Caller can pass info to callee using parameters and local variables.
- Callee can pass info to caller only through return values
- This might be too limited.

Bill Gates Net Worth

```
int iB(int iworth) {  
    iworth = iworth + 1;  
    // T2  
}
```

```
int iA() {  
    int inetWorth;  
    inetWorth = 56; //T1  
    iB(inetWorth);  
    // T3  
}
```

Bill Gates Net Worth

```
int iB(int iworth) {  
    iworth = iworth + 1;  
    // T2  
}
```

```
int iA() {  
    int inetWorth;  
    inetWorth = 56; //T1  
    iB(inetWorth);  
    // T3  
}
```

T1

iA() inetWorth 56

Bill Gates Net Worth

```
int iB(int iworth) {  
    iworth = iworth + 1;  
    // T2  
}
```

```
int iA() {  
    int inetWorth;  
    inetWorth = 56; //T1  
    iB(inetWorth);  
    // T3  
}
```

T2

iB()

iworth

57

iA()

inetWorth

56

Bill Gates Net Worth

```
int iB(int iworth) {  
    iworth = iworth + 1;  
    // T2  
}
```

```
int iA() {  
    int inetWorth;  
    inetWorth = 56; //T1  
    iB(inetWorth);  
    // T3  
}
```

T3

iA() inetWorth 56

What we want ...

T1

iA() inetWorth 56

What we want . . .

Instead of a copy, `iB()` receives a pointer to `inetWorth`. `iB()` dereferences the pointer to access and change `inetWorth`.

T2

`iB()`

`iworth`

*

`iA()`

`inetWorth`

57



What we want ...

T3

iA() inetWorth 57

What do we change to do that?

```
int iB(int iworth) {  
    iworth = iworth + 1;  
    // T2  
}
```

```
int iA() {  
    int inetWorth;  
    inetWorth = 56; //T1  
    iB(inetWorth);  
    // T3  
}
```

T3

iA() inetWorth 57

What do we change to do that?

```
int iB(int * piworth) {  
    iworth = iworth + 1;  
    // T2  
}
```

```
int iA() {  
    int inetWorth;  
    inetWorth = 56; //T1  
    iB(&inetWorth);  
    // T3  
}
```

T3

iA() inetWorth 57

What do we change to do that?

```
int B(int * worth) {  
    worth = worth + 1;  
    // T2  
}
```

```
int A() {  
    int netWorth;  
    netWorth = 56; //T1  
    B(&netWorth);  
    // T3  
}
```

T3

A()

netWorth

57

What do we change to do that?

```
int B(int * worth) {  
    worth = worth + 1;  
    // T2  
}
```

```
int A() {  
    int netWorth;  
    netWorth = 56; //T1  
    B(&netWorth);  
    // T3  
}
```

T3

A()

netWorth

57

Using pointers avoids making copies

- efficient. (e.g. arrays)
- unclear which copy is correct

A person with one watch always knows what time it is. A person with two is never sure.

Swap - you try

```
int main() {  
    int ix = 6;  
    int iy = 15;  
    swap(&ix, &iy);  
}
```

Swap - you try

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

```
int main() {  
    int ix = 6;  
    int iy = 15;  
    swap(&ix, &iy);  
}
```

<Heap Memory>

aka dynamic memory

Advantages

- lifetime: can build sth. in function and return it.
- size: can control memory allocation precisely. (e.g., estimating array sizes)

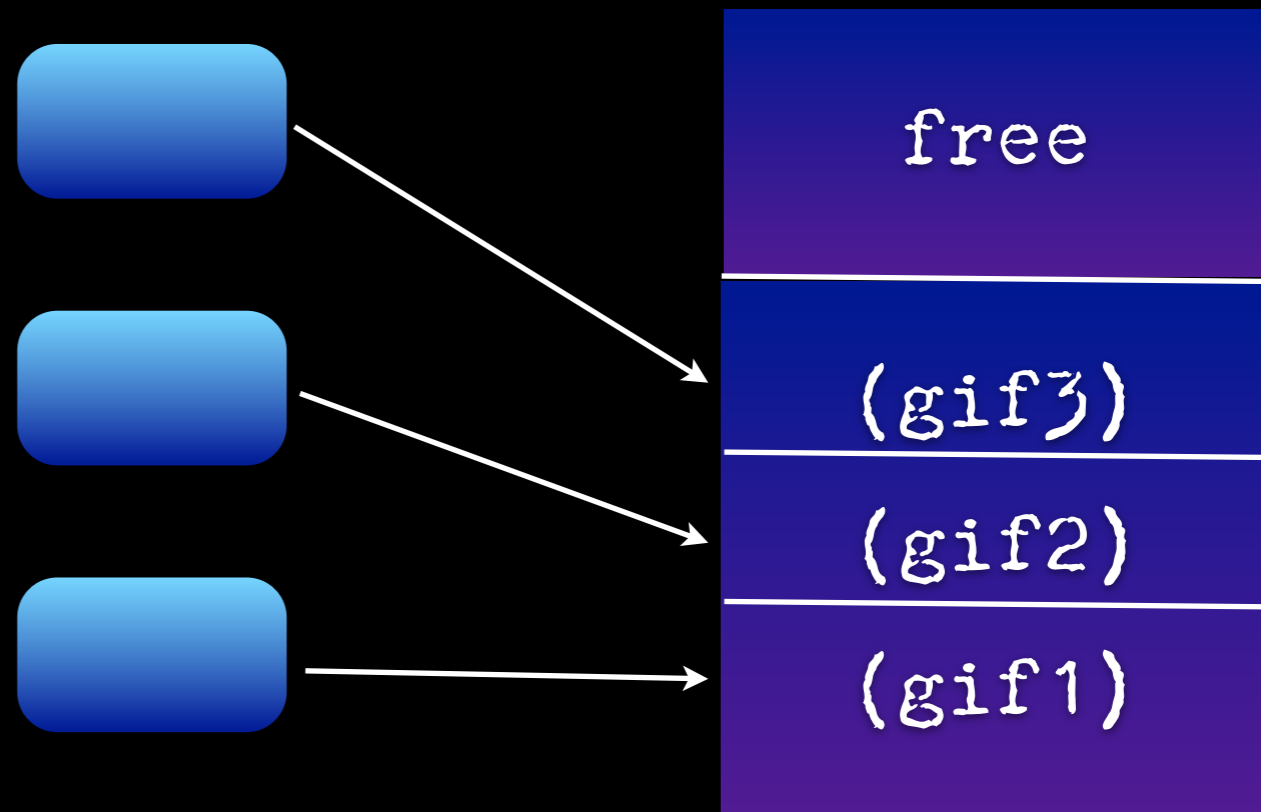
Disadvantages

- more work
- more bugs

Heap -- allocation

Local

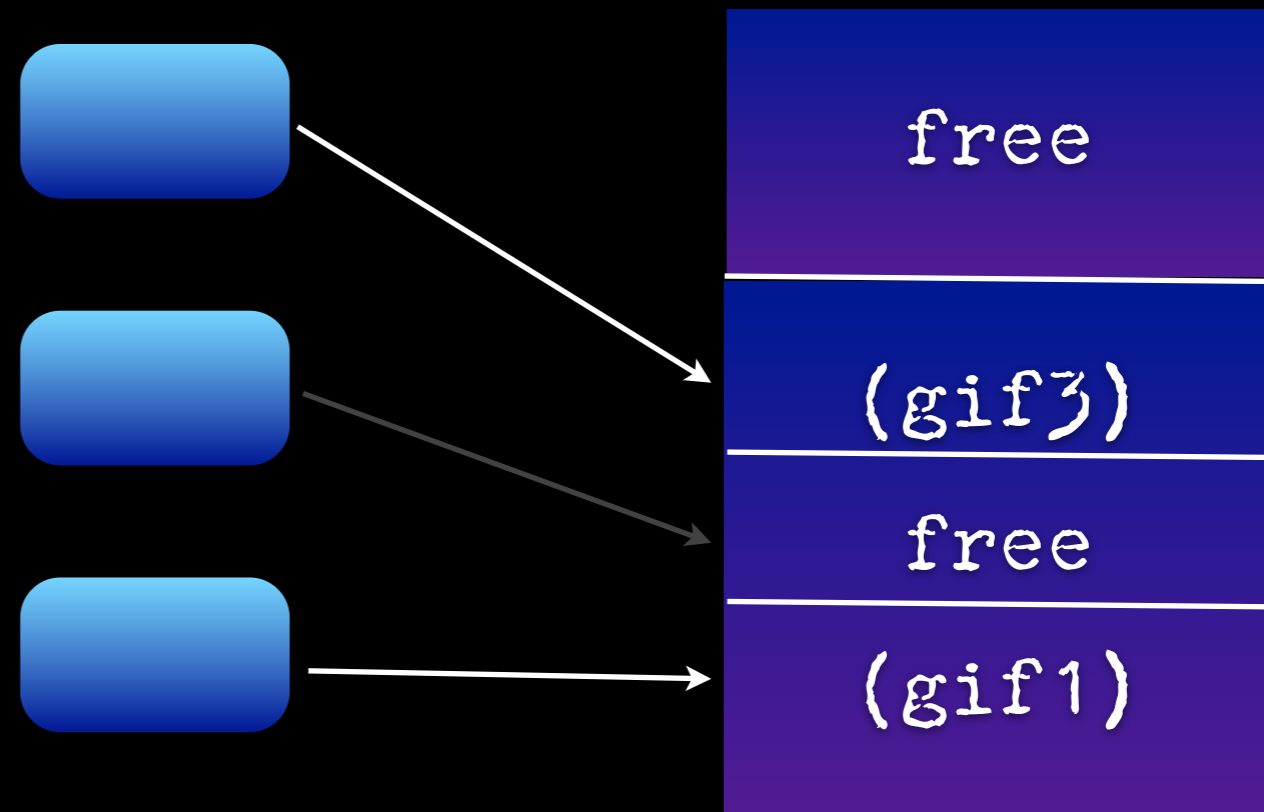
Heap



Heap -- deallocation

Local

Heap



C++ Specifics

- `new int;`
returns a pointer to a new heap block of the requested size.
it returns NULL if it cannot allocate due to heap being full.

C++ Specifics

- `delete void* heapBlockPointer` takes a pointer to a heap block and returns the block to the free pool for later reuse
- `new` and `free` should be balanced.

```
int i;  
int *pi1 = &i;  
int *pi2;  
int *pJake = new int(42);  
int *pJake2 = pJake;
```

```
delete i;           // error, i is not a pointer  
delete pi1;        // error, pi1 refers to a local  
delete pJake;      // ok  
delete pJake2;     // undefined, memory already freed  
delete pi2;        // ok, always ok to delete null pointer.
```

Simple example

```
int iHeap1() {  
    int *piFoo;  
  
}
```

Local

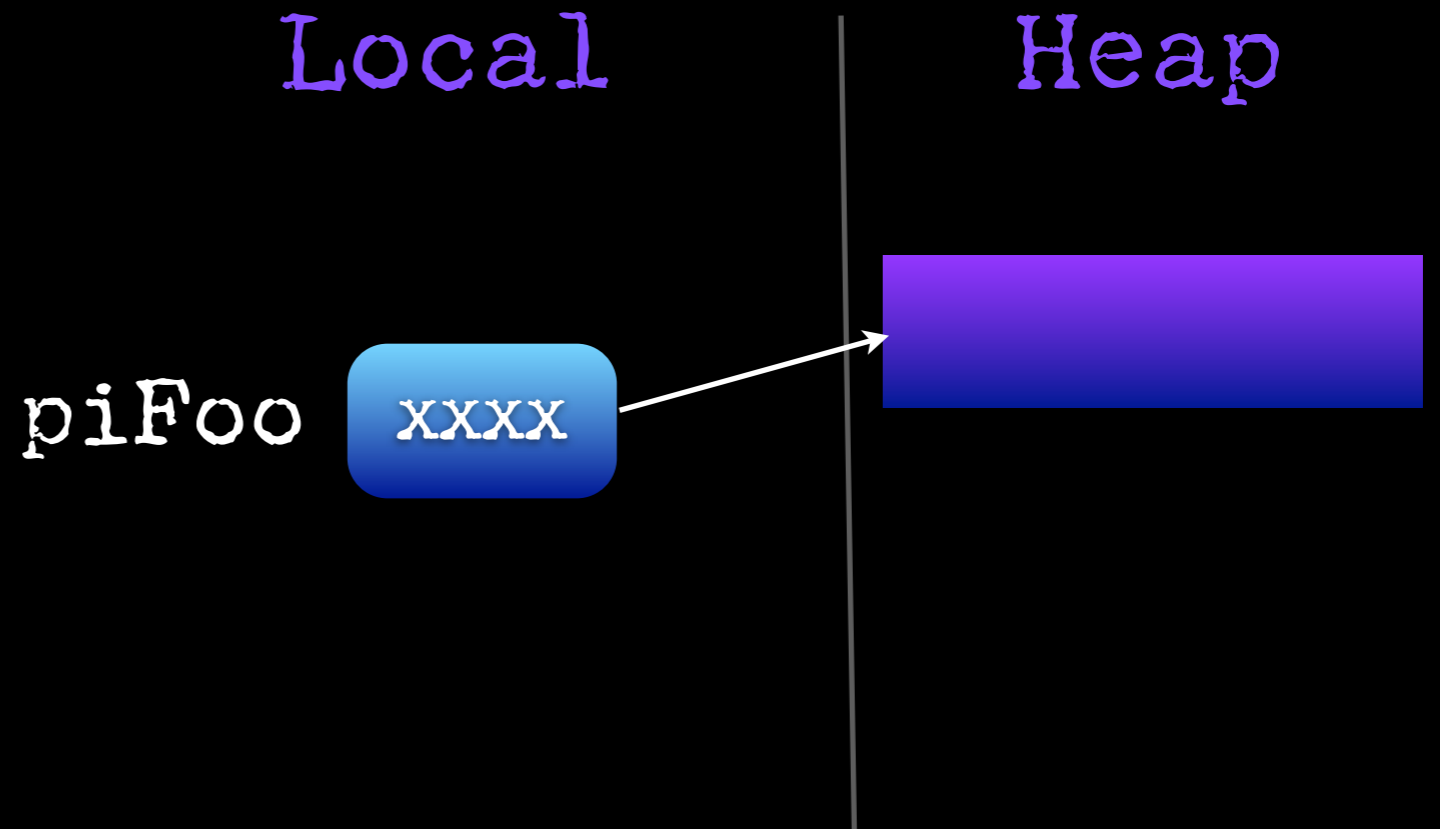
Heap

piFoo

NULL

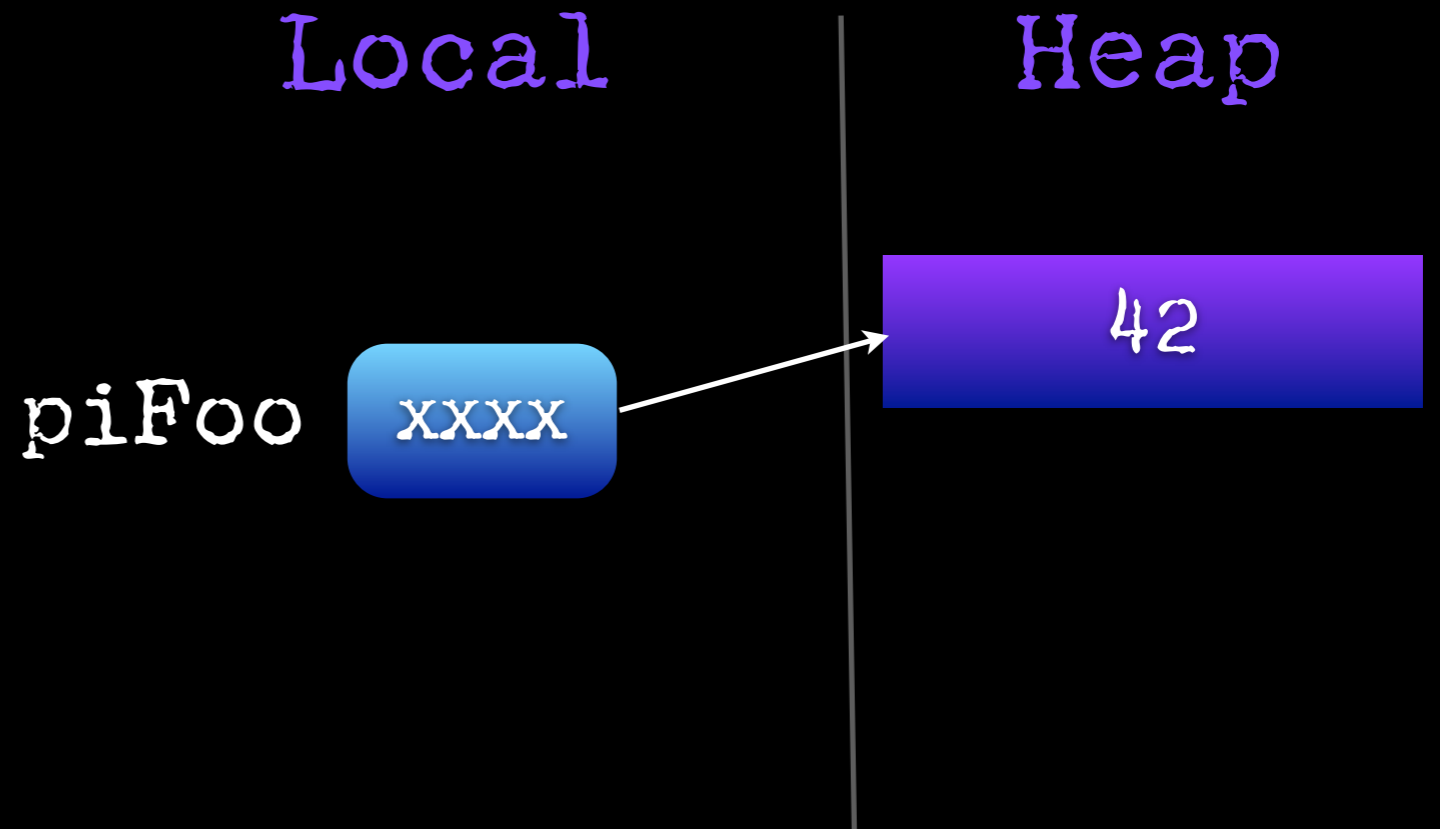
Simple example

```
int iHeap1() {  
    int *piFoo;  
    piFoo = new int;  
  
}
```



Simple example

```
int oHeap1() {  
    int *piFoo;  
    piFoo = new int;  
    *piFoo = 42;  
}
```



Compare

```
int * iCreateInt(int ix) {  
    int *piFoo;  
    piFoo = new int;  
    *piFoo = ix;  
    return piFoo;  
}
```

```
int * iCreateInt2(int ix) {  
    int itmp;  
    itmp = ix;  
    return &itmp;  
}
```

Memory leaks

- memory on the heap is allocated but never deallocated.
- For small, short-lived programs this is not a problem.
- They are a problem for programs that run an indeterminate amount of time. (for ex., OSs)

Memory leaks

- Many commercial programs have memory leaks.
- When they run for long enough they fill up the heap and crash.
- Firefox 2,

Summary: C++ pointers & memory

Create a pointer using *

```
int * piPoints;  
float * pfFrequency;
```

Allocate memory on heap using new

```
piPoints = new int;  
pfFrequency = new float;
```

Dereference a pointer using *

```
*piPoints = 42;
```

```
*pfFrequency = 440.00;
```

When done you must free
memory

```
delete piPoints;  
delete pfFrequency;
```

</ August 2011>

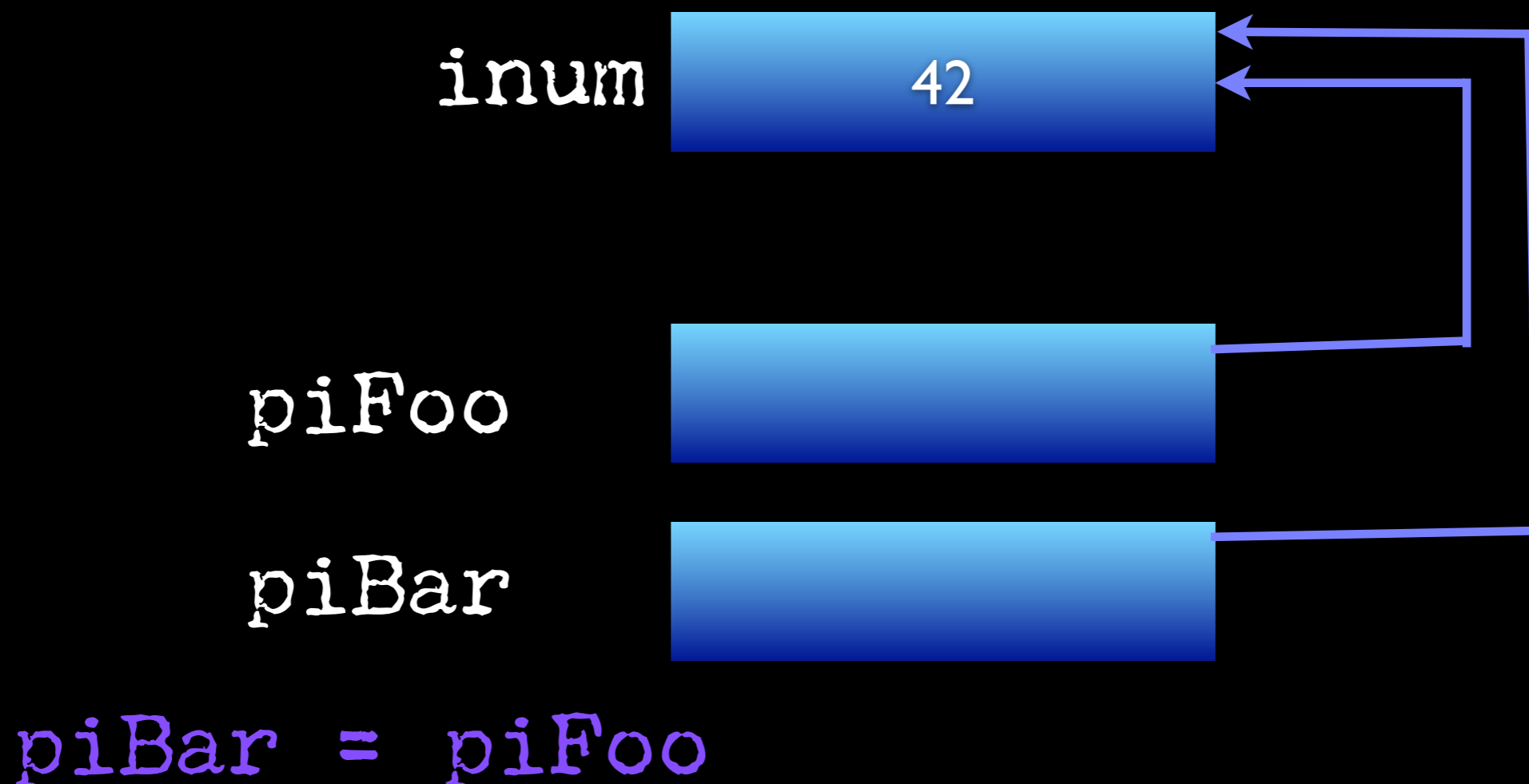
January 2012

cs230 up to this year.

<one last thing>

The magic wand of pointer assignment

- The assignment operator (=) between 2 pointers makes them point to the same pointee.



One last thing ...

```
int main()
{
    struct node * pNodeList;
    struct node * pNodeHead = NULL;
    pNodeList = pNodeMakeShortList();
    printList(pNodeList);
    pNodeAddFront(pNodeList, 77);
    printList(pNodeList);
}
```

One last thing ...

```
void pnodeAddFront(struct node * pnodeAlist, int ix)
{
    struct node * pnodetmp;
    pnodetmp = (struct node *) malloc(sizeof(struct node));
    pnodetmp->next = pnodeAlist;
    pnodetmp->data = ix;
    pnodeAlist = tmp;
    printList(pnodeAlist);
}
```

```
int main()
{
    struct node * pnodeList;
    struct node * pnodeHead = NULL;
    pnodeList = pnodeMakeShortList();
    printList(pnodeList);
    pnodeAddFront(pnodeList, 77);
    printList(pnodeList);
}
```

```
Macintosh-2:Desktop $ ./a.out
Data: 1 Next: 0x100130
Data: 2 Next: 0x100140
Data: 3 Next: 0x0
Data: 77 Next: 0x100120
Data: 1 Next: 0x100130
Data: 2 Next: 0x100140
Data: 3 Next: 0x0
WHAT IS YOUR PREDICTION?
```

One last thing . . .

```
void pnodeAddFront(struct node * pnodeAlist, int ix)
{
    struct node * pnodetmp;
    pnodetmp = (struct node *) malloc(sizeof(struct node));
    pnodetmp->next = pnodeAlist;
    pnodetmp->data = ix;
    pnodeAlist = tmp;
    printList(pnodeAlist);
}
```

```
int main()
{
    struct node * pnodeList;
    struct node * pnodeHead = NULL;
    pnodeList = pnodeMakeShortList();
    printList(pnodeList);
    pnodeAddFront(pnodeList, 77);
    printList(pnodeList);
}
```

```
Macintosh-2:Desktop $ ./a.out
Data: 1 Next: 0x100130
Data: 2 Next: 0x100140
Data: 3 Next: 0x0
Data: 77 Next: 0x100120
Data: 1 Next: 0x100130
Data: 2 Next: 0x100140
Data: 3 Next: 0x0
Data: 1 Next: 0x100130
Data: 2 Next: 0x100140
Data: 3 Next: 0x0
```

One last thing . . .

```
void pnodeAddFront(struct node ** pnodeAlist, int ix)
{
    struct node * pnodetmp;
    pnodetmp = (struct node *) malloc(sizeof(struct node));
    pnodetmp->next = pnodeAlist;
    pnodetmp->data = ix;
    pnodeAlist = tmp;
    printList(pnodeAlist);
}
```

```
int main()
{
    struct node * pnodeList;
    struct node * pnodeHead = NULL;
    pnodeList = pnodeMakeShortList();
    printList(pnodeList);
    pnodeAddFront(&pnodeList, 77);
    printList(pnodeList);
}
```

```
Macintosh-2:Desktop $ ./a.out
Data: 1 Next: 0x100130
Data: 2 Next: 0x100140
Data: 3 Next: 0x0
Data: 77 Next: 0x100120
Data: 1 Next: 0x100130
Data: 2 Next: 0x100140
Data: 3 Next: 0x0
Data: 1 Next: 0x100130
Data: 2 Next: 0x100140
Data: 3 Next: 0x0
```