

C++ for Java Programmers – II

WL101.Ib

©Wilhansen Li 2010 <wil@nohakostudios.net>

Risky but rewarding.

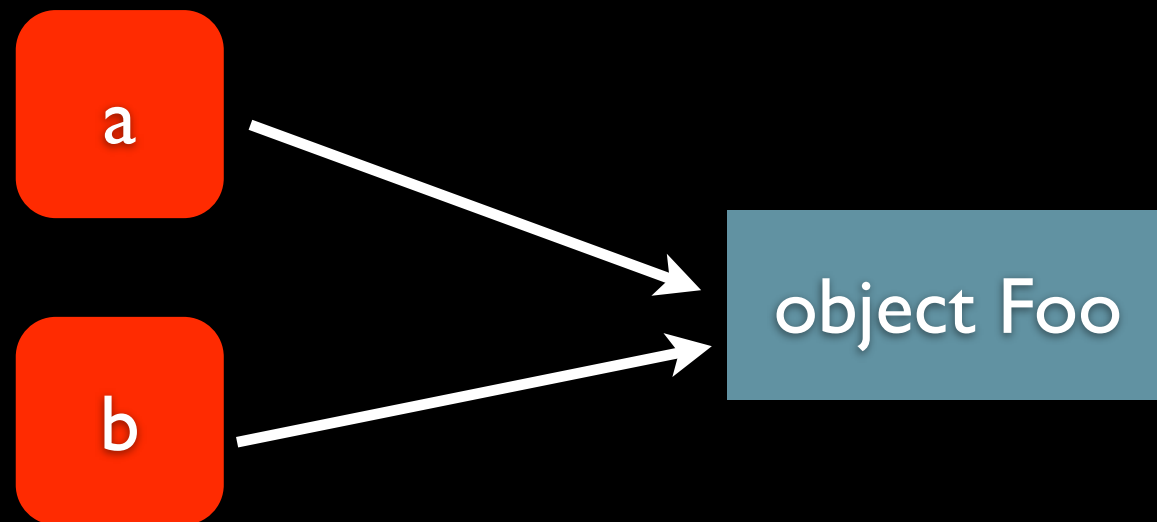
Outline

- Pointers
- References
 - ✦ `sizeof`
- Arrays
 - ✦ Strings
- Memory Model
 - ✦ Dynamic Allocation
 - ✦ L-values and R-values
 - ✦ Scoping and Lifetime
 - ✦ Casts
 - ✦ Memory layout

Pointers

- Recall: In Java, all object identifiers are pointers.

```
public static void main(String [] args) {  
    Foo a = new Foo();  
    Foo b = a;  
}
```



Pointers

- In C++, this is not the case.
- Remember the principle: *Treat all objects as if they are primitives.*

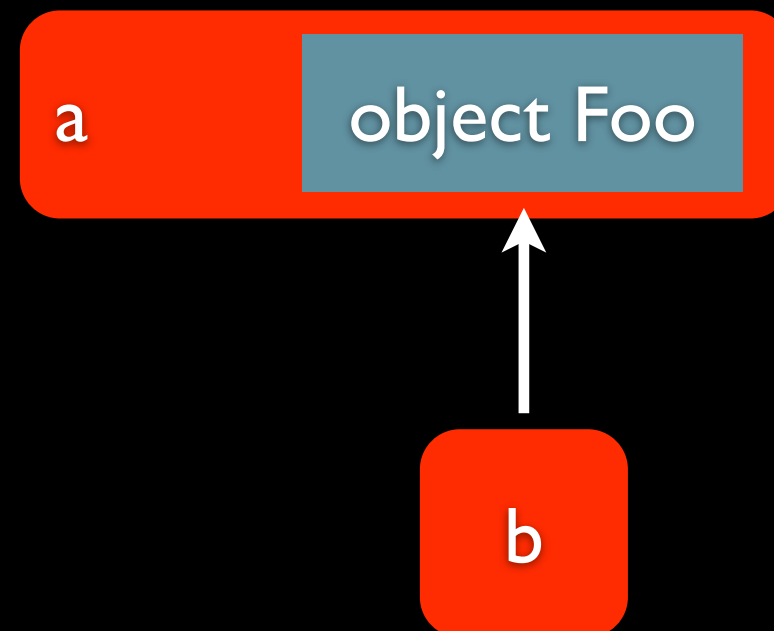
```
int main() {  
    Foo a;  
    Foo b = a;  
    return 0;  
}
```



Pointers

- Such pointers have to be declared explicitly using `*` when declaring variables.
- The “address” of a variable can be retrieved by the address-of operator `&`.
- Everything else works like Java... *almost*.

```
Foo a;  
Foo *b = &a;
```



Pointers

- Adhering again to the C++ type principle, you can do the same for primitives.
- No `int` v.s. `Integer`, `float` v.s. `Float`, `char` v.s. `Character`, `boolean` v.s. `Boolean`, etc.

```
int main() {  
    int a = 0;  
    int *b = &a;  
    return 0;  
}
```

Pointers

- The *** modifier only affects the immediate identifier. Compound declarations beware!
- It doesn't matter where the space is placed.
- Typedef may help here.

```
int *a, b; //b is not a pointer!  
typedef int* IntPtr;  
IntPtr c, d; //c and d are both pointers.
```

Pointers

- Temporary objects, known as **R-values**, have no addresses thus, they can't be pointed to.

```
int *a = &(1 + 2); //illegal!
```

Generally, if the value has no name, it has no address (more to this later).

Pointers

- A “null pointer” in C++ is basically **0**.
- Some people (and compilers) predefine **NULL** and use it instead.
- C++0x defines **nullptr** to fill this gap.

```
int main() {  
    Foo *a = 0;  
    Foo *b = NULL; //works most of the time.  
    return 0;  
}
```

Pointers

- Given a pointer, we access the actual object using the “dereference” *** operator.
- It can be used both for assigning and reading.

```
int main() {  
    int a = 0;  
    int *b = &a;  
    *b = 0xBEEF;  
    return a; //what does this return?  
}
```

const Pointers

- Pointers can be **const** in 2 non-exclusive senses:
 - The location it's pointing to is **const** (**a**).
 - The value of the pointer itself is **const** (**b**).
- They can be both (**c**).

```
const int *a;  
int * const b;  
const int * const c;
```

Read from *right to left*.

Const Pointers

- Note that dereferenced const pointers cannot be assigned even if the location they're pointing to is non-const.

```
int a = 5;  
const int *ptr = &a;  
*ptr = 10; //will not compile!
```

Pointers

- For objects, operator `->` is a shortcut to `(*object).member`

```
foo->bar
```



```
(*foo).bar;
```

Checkpoint 2.0

Assume class `Foo` has an `int` field named `bar`,
what does `main` return?

What happens when `b->bar = 20;` is removed?

```
int main() {  
    Foo a;  
    Foo *b = &a;  
    a.bar = 10;  
    b->bar = 20;  
    return (*b).bar;  
}
```

Checkpoint 2.1

What does `swap` do and what does `main` return?

```
void swap(int *a, int *b) {  
    int temp = *a;  
    *a = *b;  
    *b = temp;  
}  
  
int main() {  
    int a = 0, b = 1;  
    swap(&a, &b);  
    return a; //what does this return?  
}
```

Bonus: What's the equivalent in Java?

Checkpoint 2.2

What does each version do? Are they all correct?

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

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

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

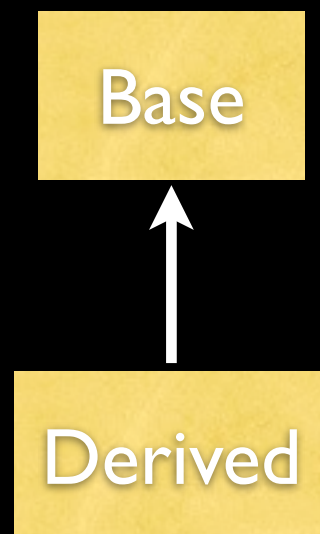

Checkpoint 2.3

What's the improvement of this over the earlier version?

```
void swap(int *a, int *b) {  
    if ( !a || !b ) return;  
    int temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

Pointer Polymorphism

- Pointers and references (later) are the only ways to do polymorphism.
- Like Java, no explicit cast is needed when upcasting.



```
Derived d;  
Base *b = &d;
```

Checkpoint 2.4

What's possibly wrong with the following?

```
void foo(Base b) {  
    //do something to b  
}  
  
int main() {  
    Derived d;  
    foo(d);  
    return 0;  
}
```

Checkpoint 2.4 “solution”

Correct version:

```
void foo(Base *b) {  
    //do something to b  
}  
  
int main() {  
    Derived d;  
    foo(&d);  
    return 0;  
}
```

Void Pointer

- The void pointer (`void *`) is the “mother of all pointers”; any pointer can be upcasted to a void pointer.
- Downcasts are necessary when dereferencing.
- Similar to `Object` in Java.

```
int main() {  
    int a = 0;  
    Foo b;  
    void *c;  
    c = &b;  
    c = &a;  
    return *((int*)c);  
}
```

Function Pointers

- Pointers can also refer to functions.
- Declaration is in the form:

```
return_type (*PointerName) (parameters);
```

1. Never omit the first set of parenthesis.
2. Typedef will greatly help.
3. Parameter names can be omitted.
4. They are invoked as if they are normal functions.

Function Pointer Example

```
int* a();  
int (*b)();  
int* (*c)();  
void (*d)(int);  
void (*e)(void (*)());
```

```
typedef int (*IntFP)();
```

```
IntFP f;  
IntFP (*g)(IntFP);
```

Function Pointer Example

```
#include<iostream>
using namespace std;

int add(int a, int b) { return a + b; }
int multiply(int a, int b) { return a * b; }

★ typedef int (*binary_op)(int, int);

int main() {
    ★ binary_op operation = 0;
    int x, y;
    char op;
    cin >> x >> op >> y;
    if ( op == '+' ) {
        ★ operation = add;
    } else {
        ★ operation = multiply;
    }
    ★ cout << operation(x, y) << '\n';
    return 0;
}
```


Checkpoint 2.5

Declare a function pointer that...

- returns a void pointer.
- accepts a pointer to an int.

Describe the following:

```
void (*ptr) ( void (*) (int) );
```

Checkpoint 2.5

Declare a function pointer that...

- returns a void pointer.
- accepts a pointer to an int.

```
void* (*FunPtr)(int*);
```

Describe the following:

```
void (*ptr)( void (*) (int) );
```

Pointer to Pointers

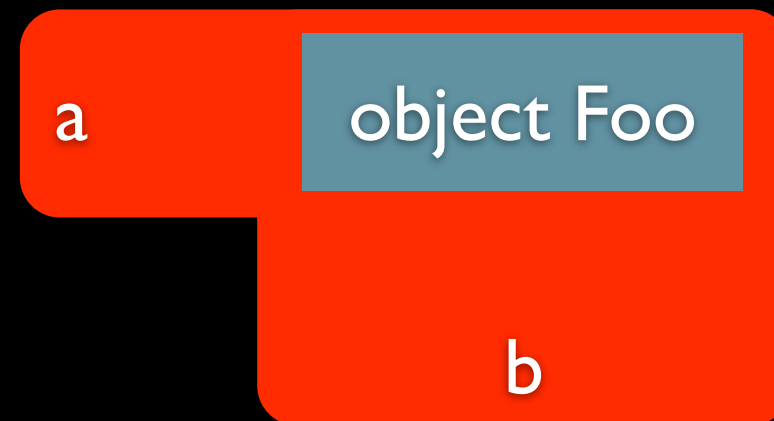
- It's also possible to declare a pointer of pointers.
- Used in some C programs but not so much in C++.

```
int **ptrptr;
```

References

- References are similar to pointers that they are aliases.
- However, unlike pointers, they're “permanent” – they can't be reassigned to point to another object.
- Declare them by using the **&** modifier.
- They have to be bound upon declaration.

```
int main() {  
    Foo a;  
    Foo &b = a;  
    return 0;  
}
```



References

- Unlike pointers, they are transparent – they can be used like normal variables.

```
int main() {  
    int a = 0;  
    int &b = a;  
    b = 0xBEEF;  
    return a; //what does this return?  
}
```

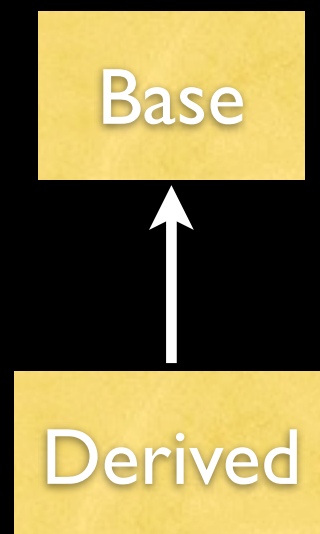
Returning References

- Functions can return references by appending `&` to their return type.
- Make sure the object is still alive when returning its reference.

```
int a = 1;
int& getFoo() {
    return a;
}
int main() {
    getFoo() = 0;
    return a;
}
```

Reference Polymorphism

- References (aside from pointers) can also be used to do polymorphism.
- No copying is involved.



```
Derived d;  
Base &b = d;
```

Reference Example

```
void swap(int &a, int &b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}  
  
int main() {  
    int a = 0, b = 1;  
    swap(a, b);  
    return a; //what does this return?  
}
```


Reference Example Comparison

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

```
int main() {  
    int a = 0, b = 1;  
    swap(a, b);  
    return a;  
}
```

Reference Version

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

```
int main() {  
    int a = 0, b = 1;  
    swap(&a, &b);  
    return a;  
}
```

Pointer Version

References v.s. Pointers

- Pointers are NULL-able, references aren't.

```
int *a = NULL;  
int &b = NULL; //no!
```

- Pointers are reassignable, references aren't (they are bound upon initialization).

```
int c;  
a = &c;  
&b = c; //no!
```

References v.s. Pointers

- As a general guideline, use references when you can, pointers when needed.
- References are usually used in short-term execution (function parameters).
- Pointers are used for long-term usage like in data structures (trees, linked-list, etc.)
- Case in point: the `swap` function.

References + Const

- Sometimes, you just want to pass “heavy” objects (like vectors) and not modify it.
- Though consts help, they still copy objects.
- Const + References = const references!

```
void printArray(const string str) {  
    for ( size_t i = 0; i < str.size(); ++i )  
        cout << str[i] << ' ';  
}
```



```
void printArr(const string &str) {  
    for ( size_t i = 0; i < str.size(); ++i )  
        cout << str[i] << ' ';  
}
```

References + Const

- Also, const references are more “compatible”

```
const string TEMP_DIR = "C:\\Temp";
```

```
void foo(string &path);
```

```
void bar(const string &path);
```

```
foo(TEMP_DIR);
```

```
bar(TEMP_DIR);
```

Which will fail to compile?

Reference + Const

Important: Const references
can also accept R-values.

```
int main() {  
    int &a = 1 + 1; //cannot  
    const int &b = 1 + 1; //can  
    return 0;  
}
```

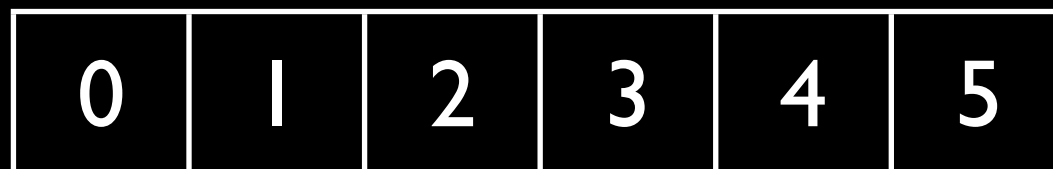
Alias Usage Summary

- References and pointers are usually used with functions on two occasions:
 1. Huge objects are being passed (they incur copying overhead)
 2. Return values are complex (like the `swap` function)
 3. Parameters need to be modified.
- As a rule-of-the-thumb, always pass objects *by (const) reference* for efficiency.

BREAKT13M

Arrays

- C++ has two kinds of built-in arrays: static and dynamic.
- Array sizes have to be manually tracked.
- No “ArrayIndexOutOfBoundsException” exception is thrown when you go out of bounds.
- Important: Arrays are always contiguous.



Static Arrays

- Sizes are fixed at compile time.
- Like normal variables, initial content is dirty.
- Only literals or `const` can be used.
- Array size must be tracked manually (i.e. no `Array.length` property).

```
int arr[10];  
const int ARRAY_SIZE = 20;  
int arr2[ARRAY_SIZE];  
for ( int i = 0; i < ARRAY_SIZE; ++i )  
    arr2[i] = i;
```

Static Arrays

- Notice the difference in the `[]` placement.

```
//Java
```

```
int [] arr = new int[10];
```

```
//C++
```

```
int arr[10];
```

Static Arrays

- Contents can be specified upon declaration.
- When doing so, the size may be omitted.
- If declaration content is less than the array size, the rest of the entires are zeroed out.

```
int a[2] = {0, 1};  
int b[] = {0, 1, 2, 3};  
int c[10] = {0};
```

Dynamic Arrays

- Pointers must be used.
- Create with `new`, release with `delete []`
- Since C++ has no GC, anything dynamically allocated must be explicitly released.
- As with static arrays, sizes are manually tracked.

```
int *a = new int[10];  
delete [] a;
```

Dynamic Array

- Unlike in Java, allocated dynamic arrays are already initialized.
- If the Java behavior is preferred, use a pointer to a pointer. (later)

Java

```
Foo [] a = new Foo[2];  
a[0] = new Foo();  
a[1] = new Foo();
```

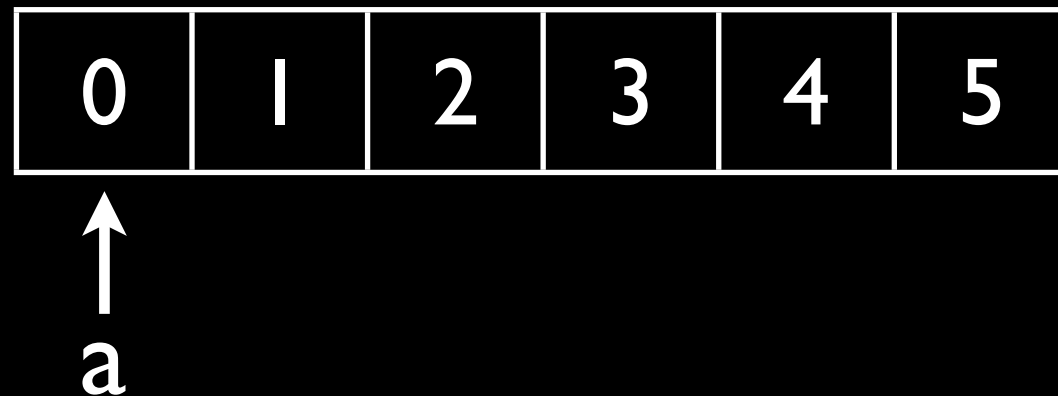
C++

```
Foo *a = new Foo[2];  
//a[0] and a[1] already constructed
```

Pointer-Array Dualism

- Recall the array layout.
- Dynamic arrays can be viewed as...

```
int *a = new int[6];
```

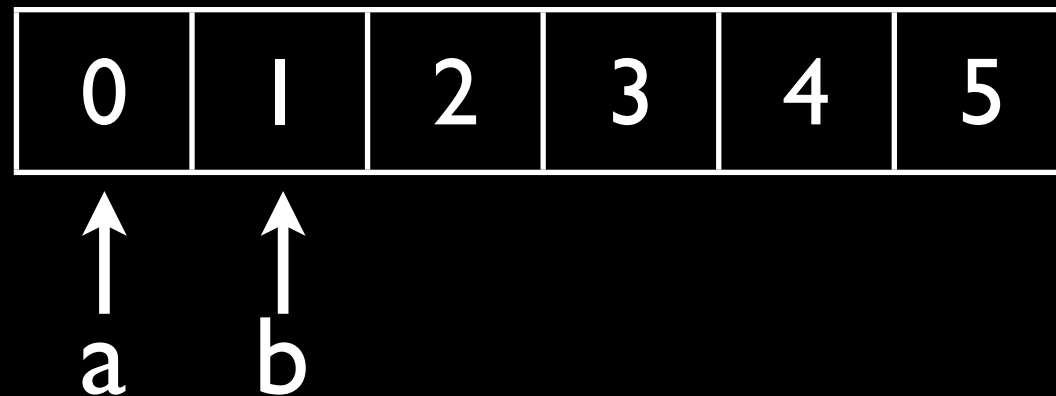


This means `a[0] == *a`

Pointer-Array Dualism

- What if I did this:

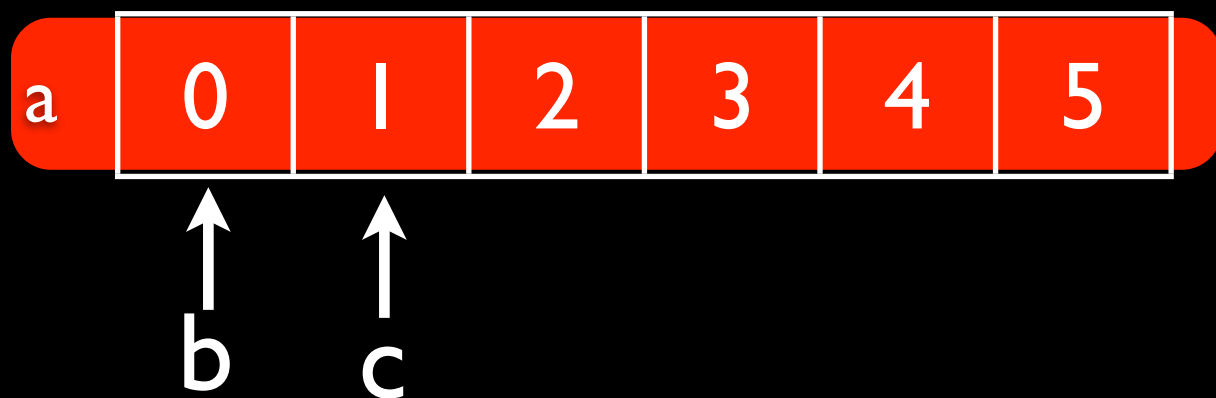
```
int *b = &(a[1]);
```



```
b[0] == a[1]
```


Pointer-Array Dualism

- What about static arrays?
- `a` is basically a pointer to the first element.
- The same concepts can be used.



```
int a[6];  
int *b = a;  
int *c = &(a[1]);
```

Passing Arrays as Arguments

- When passing arrays to functions, two pieces of information are needed:
 - Pointer to the array
 - Array size

```
int sum(const int *elements, const int n) {  
    int s = 0;  
    for ( int i = 0; i < n; ++i )  
        s += elements[i];  
    return s;  
}
```

std::vector<>

- An easier and safer way to do dynamic arrays.
- Analogous to `java.util.ArrayList`
- Access elements using `[]`.
- Automatically deallocates upon scope exit.

```
{  
    std::vector<int> arr;  
    arr.push_back(1);  
}  
//arr automatically gets released
```

std::vector<>

- Guaranteed to be contiguous.
- Can act like a normal dynamic array.

```
int sum(const int *elements, const int n) {  
    int s = 0;  
    for ( int i = 0; i < n; ++i )  
        s += elements[i];  
    return s;  
}
```

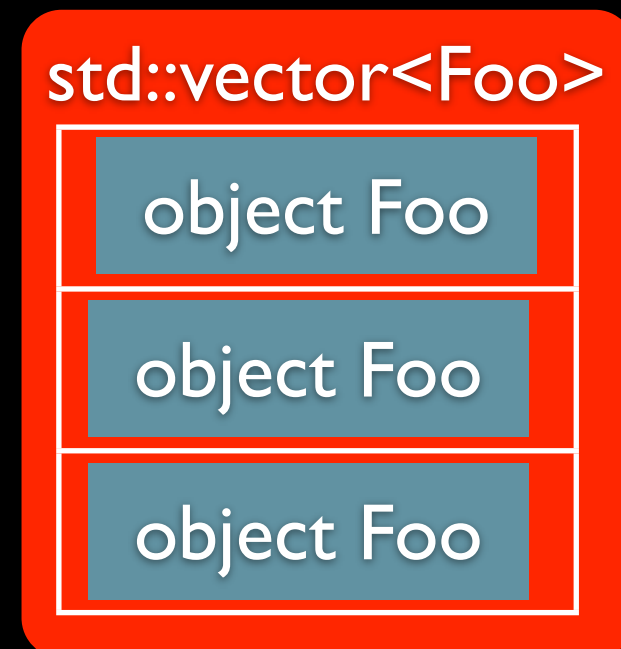
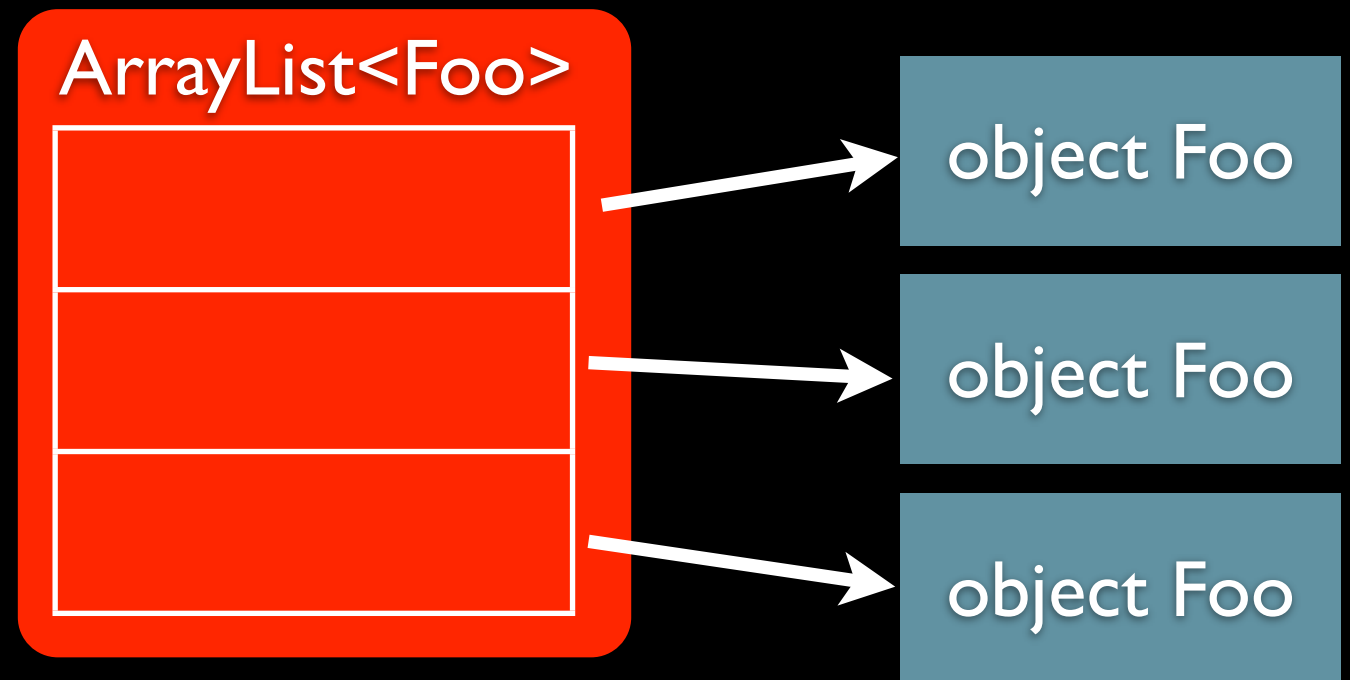
```
vector<int> a;  
//insert elements to a.  
int total = sum(&a[0]), a.size());
```

std::vector<>

- Append elements using `vector<>.push_back(item)`
- Get the size using `vector<>.size()`
- Check if it's empty by calling `vector<>.empty()` (or just check if `vector<>.size() == 0`)
- `vector<>.front()` and `vector<>.back()` to access the first and last element (will crash if the vector is empty).
- Resize using `vector<>.resize(newsize, fill)`
- Clear the vector by calling `vector<>.clear()`

std::vector<>

- Stores exactly what's in its type (not a pointer, unlike Java).
- If objects are large, it may be more efficient to store pointers of them instead; like the Java way. (later)



std::vector<> examples

```
using namespace std;
vector<int> a;
vector<float> b;
for ( int i = 0; i < 10; ++i )
    a.push_back(i);
for ( int i = 0; i < a.size(); ++i )
    b.push_back(a[i] * 1.5f);

a.clear();
```

Checkpoint 2.6

Rewrite the following to use `std::vector<>` directly.

```
int sum(const int *elements, const int n) {  
    int s = 0;  
    for ( int i = 0; i < n; ++i )  
        s += elements[i];  
    return s;  
}
```

```
int sum(const std::vector<int> &elements) {  
    int s = 0;  
    for ( int i = 0; i < elements.size(); ++i )  
        s += elements[i];  
    return s;  
}
```


Checkpoint 2.7

Is the following safe?

```
int sum(const int *elements, const int n) {  
    int s = 0;  
    for ( int i = 0; i < n; ++i )  
        s += elements[i];  
    return s;  
}
```

```
vector<int> a;  
a.push_back(0);  
a.push_back(1);
```

```
int *p = &(a[0]);  
int n = a.size();  
a.clear();
```

```
int total = sum(p, n);
```

Arrays

- I skipped the part on how to arrays having 2 dimensions and above.

BREAKT13M

Memory Model

- Scoping and Lifetime
- Casts
- Memory layout
- `sizeof`
- Strings
- Dynamic Allocation
- L-values and R-values

Scoping and Lifetime

- Every object has a lifetime subject to their scope.
 - Begins upon declaration.
 - Ends upon scope exit.
- Global and static variables die upon program exit.

```
void foo(int *a) {  
    int b;  
}  
int c;  
foo(&c);
```

Scoping and Lifetime Exercise

Copy the following code to the top of the source.

```
class Foo {  
    int i;  
public:  
    Foo(int i) : i(i) {  
        std::cout << "Foo " << i << " created\n";  
    }  
    Foo(const Foo &f) {  
        std::cout << "Foo " << f.i << " being copied\n";  
        i = f.i;  
    }  
    ~Foo() {  
        std::cout << "Foo " << i << " releasing\n";  
    }  
};
```

Scoping and Lifetime Exercise

Try to following codes and see what's the output.

a

```
Foo a(0);

int main() {
    Foo b(1);
    return 0;
}
```

b

```
int main() {
    Foo a(0), b(1);
    return 0;
}
```

c

```
int main() {
    {
        Foo a(0);
    }
    Foo b(1);
    return 0;
}
```

d

```
int main() {
    Foo a(0);
    Foo b(a);
    return 0;
}
```

f

```
void bar() {
    Foo b(1);
}

int main() {
    bar();
    Foo a(0);
    bar();
    return 0;
}
```

g

```
void bar(const Foo &p) {
    Foo b(1);
}

int main() {
    Foo a(0);
    bar(a);
    return 0;
}
```

h

```
void bar(Foo p) {
    Foo b(1);
}

int main() {
    Foo a(0);
    bar(a);
    return 0;
}
```

Scoping and Lifetime Exercise

More things to try.

i

```
Foo bar() {  
    return Foo(1);  
}  
  
int main() {  
    Foo a = bar();  
    return 0;  
}
```

j

```
Foo bar(const Foo &p) {  
    return p;  
}  
  
int main() {  
    Foo a(0);  
    Foo b = bar(a);  
    return 0;  
}
```

k

```
Foo bar() {  
    return Foo(0);  
}  
  
int main() {  
    const Foo &a = bar();  
    Foo b(1);  
    return 0;  
}
```


Checkpoint 2.8

What's wrong with this code?

```
Foo* getFoo() {  
    Foo a;  
    return &a;  
}
```

Checkpoint 2.9

Is this ok?

```
Foo* getFoo() {  
    static Foo a;  
    return &a;  
}
```

Casts

- C++ is a weakly-typed language.
- Usually, anything can be cast to anything.
- There are 2 types of casts. They both achieve the same thing.
 - C-style: Simpler but less verbose.
 - C++-style: More complex but “safer”.

C++ Casts

- C++ casts are more discriminate as to what kind of cast is used.
- 4 types:
 - `static_cast`
 - `reinterpret_cast`
 - `const_cast`
 - `dynamic_cast`

static_cast

- The most “normal” cast.
- What you’ll want/need most of the time.

```
int i = static_cast<int>(2.3/1.4);
```

- It’s “static” because it’s evaluated compile-time.

static_cast

- Cannot be used for casting across “weirder” type boundaries (casting a pointer to an integer). A compiler error will happen.
- ***Do not use*** for downcasting unless you’re 100% sure that the object type is correct or else an undefined behavior will happen.

```
Base *base = Factory::createRandomInstance();  
Derived *d = static_cast<Derived*>(base); //dangerous!
```

reinterpret_cast

- Used for “weirder” casts that require reinterpretations.
- Examples
 - `int*` to `char*`
 - `int*` to `int`
 - `float*` to `char*`

```
char *stuff = "Hello";  
int *p = reinterpret_cast<int*> (stuff);
```

const_cast

- Used to cast-away const.
- Not used often.

dynamic_cast

- Used for downcasting.
- When the downcast cannot be performed, `0` (null pointer) is returned.
- A bit slower than `static_cast` due to type checking at runtime (“dynamic”).

```
Base *base = Factory::createRandomInstance();  
Derived *d = dynamic_cast<Derived*>(base); //null when cannot
```

C casts

- Syntax exactly like Java's.
- `static_cast`, `reinterpret_cast` and `const_cast` combined (but no `dynamic_cast`!)
- Normal form

```
float f = (float)3/5;
```

- Function form

```
float f = float(3)/5;
```

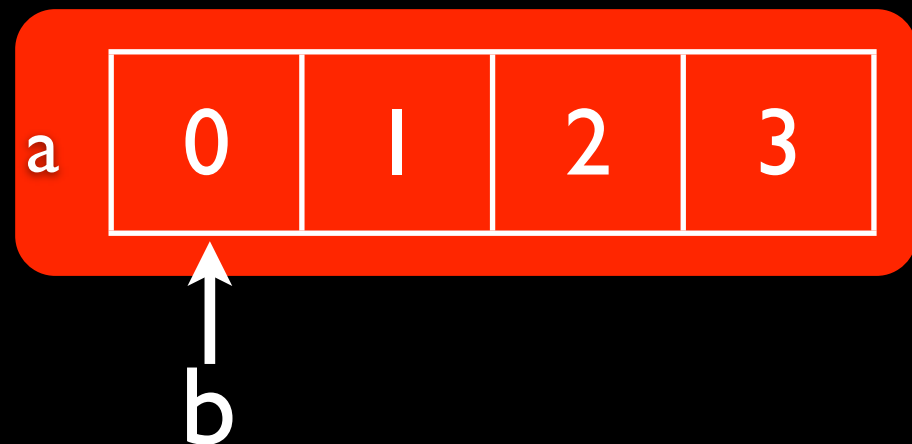
Memory Layout

- C++, being a mid-level language, exposes some aspects of the hardware.
- Disclaimer: The following examples are for *illustrative purposes only*.
- **Do not** do them unless you know and are **really** sure what you're doing.
- They are to give an insight on how computers handle programs.

Memory Layout

- An `int` has 4 bytes.
- Each byte can be accessed by turning it into an array of bytes (`char`).

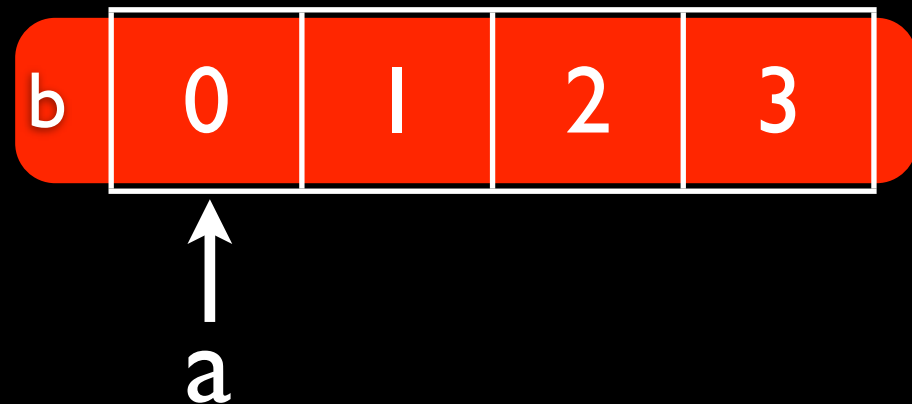
```
int a;  
char *b = (char*)&a;
```



Memory Layout

- Interpreting an array of bytes as an `int` can also be done.

```
char b[] = {0, 1, 2, 3};  
int *a = (int*)b;
```



Memory Layout

- A pointer is essentially a number representing where in the memory is the value.
- A pointer in a 32-bit OS is 32-bits (`int`) while a pointer in a 64-bit OS is 64-bits (`long long`).
- You can `#include <stdint.h>` and use the typedef `uintptr_t`.
- The address can be printed by `reinterpret_cast` to the appropriate type.

Example: Print the value of a pointer.

```
#include <cstdint>
#include <iostream>
using namespace std;

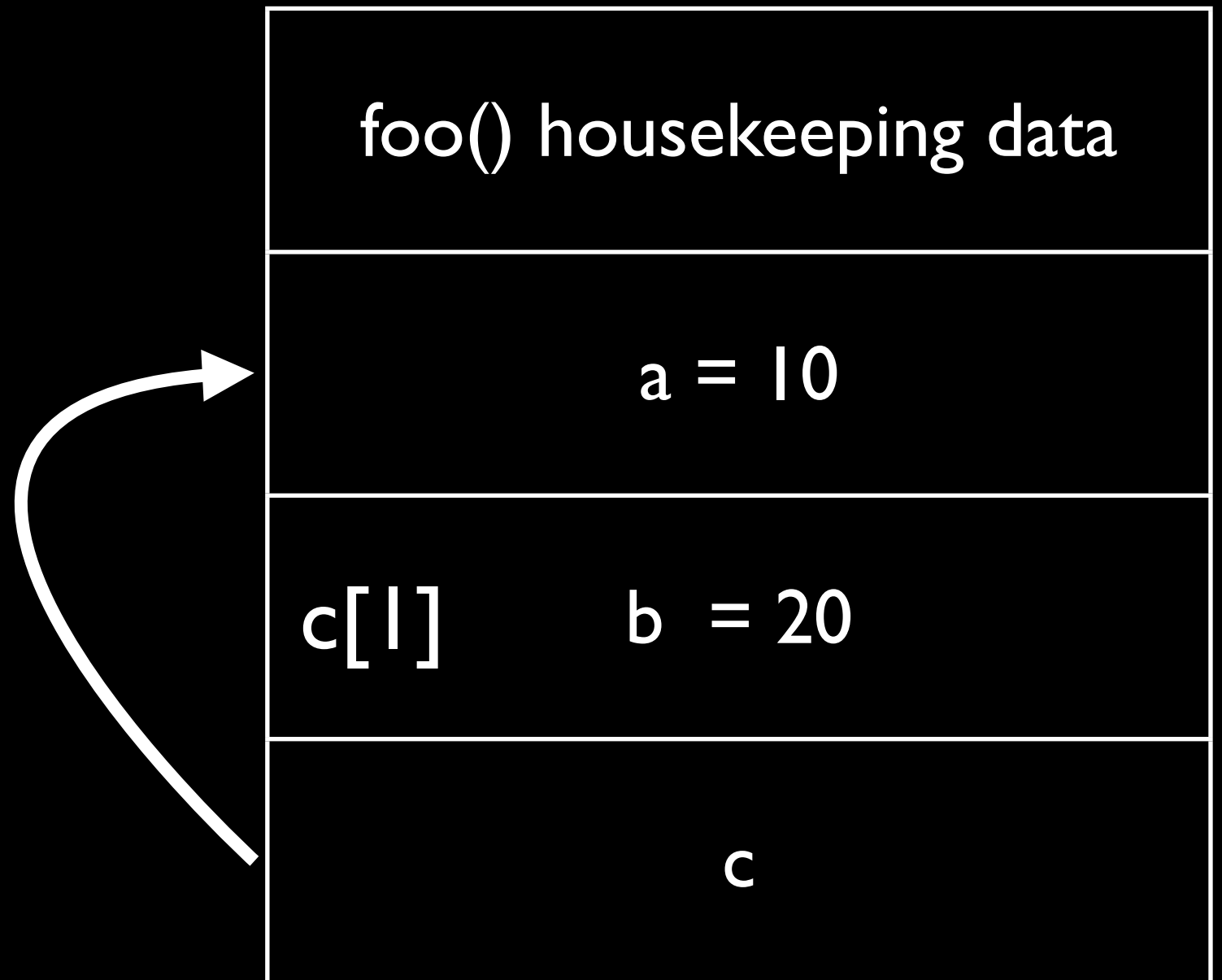
int main() {
    uintptr_t mainPtr = reinterpret_cast<uintptr_t>(main);
    cout << "Address of main: " << mainPtr << endl;
    return 0;
}
```

Memory Layout

- Weird things can happen if your array runs out of bounds.
- Generally they're undefined but some cases can be isolated.
- Disclaimer: Never **ever** rely on these behaviors. They may help in debugging but don't intentionally cause them.

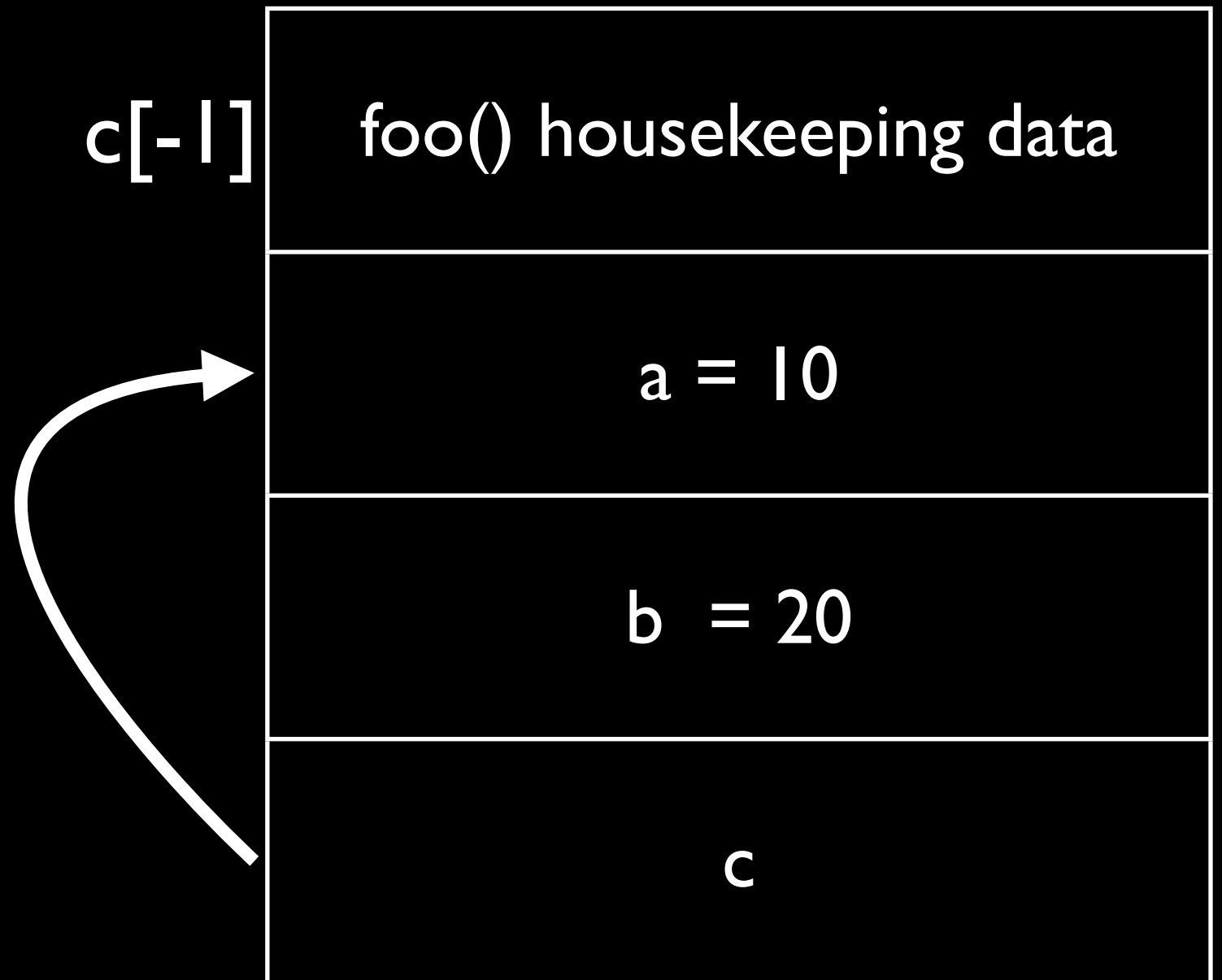
Case 0:

```
int foo() {  
    int a = 10;  
    int b = 20;  
    int *c = &a;  
    return c[1];  
}
```



Case I:

```
int foo() {  
    int a = 10;  
    int b = 20;  
    int *c = &a;  
    return c[-1];  
}
```



Will return the internal data of the function.
Modifying this will result in a crash.

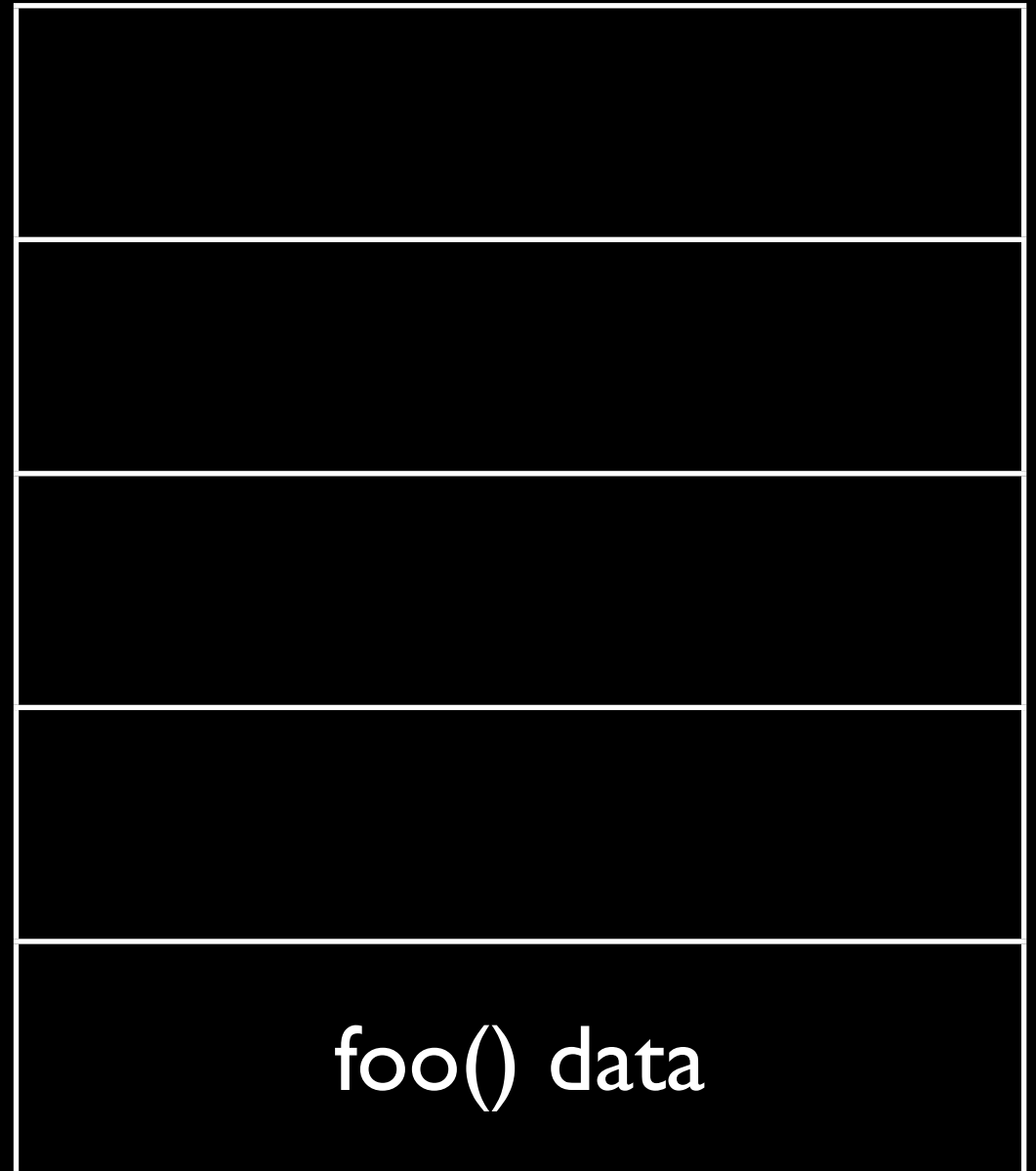
Case 2:

```
int bar() {  
    int b_a;  
}  
  
int foo() {  
    int f_a;  
    bar();  
    {  
        int f_b;  
    }  
    int f_c;  
}
```


Suppose `foo()` is called.

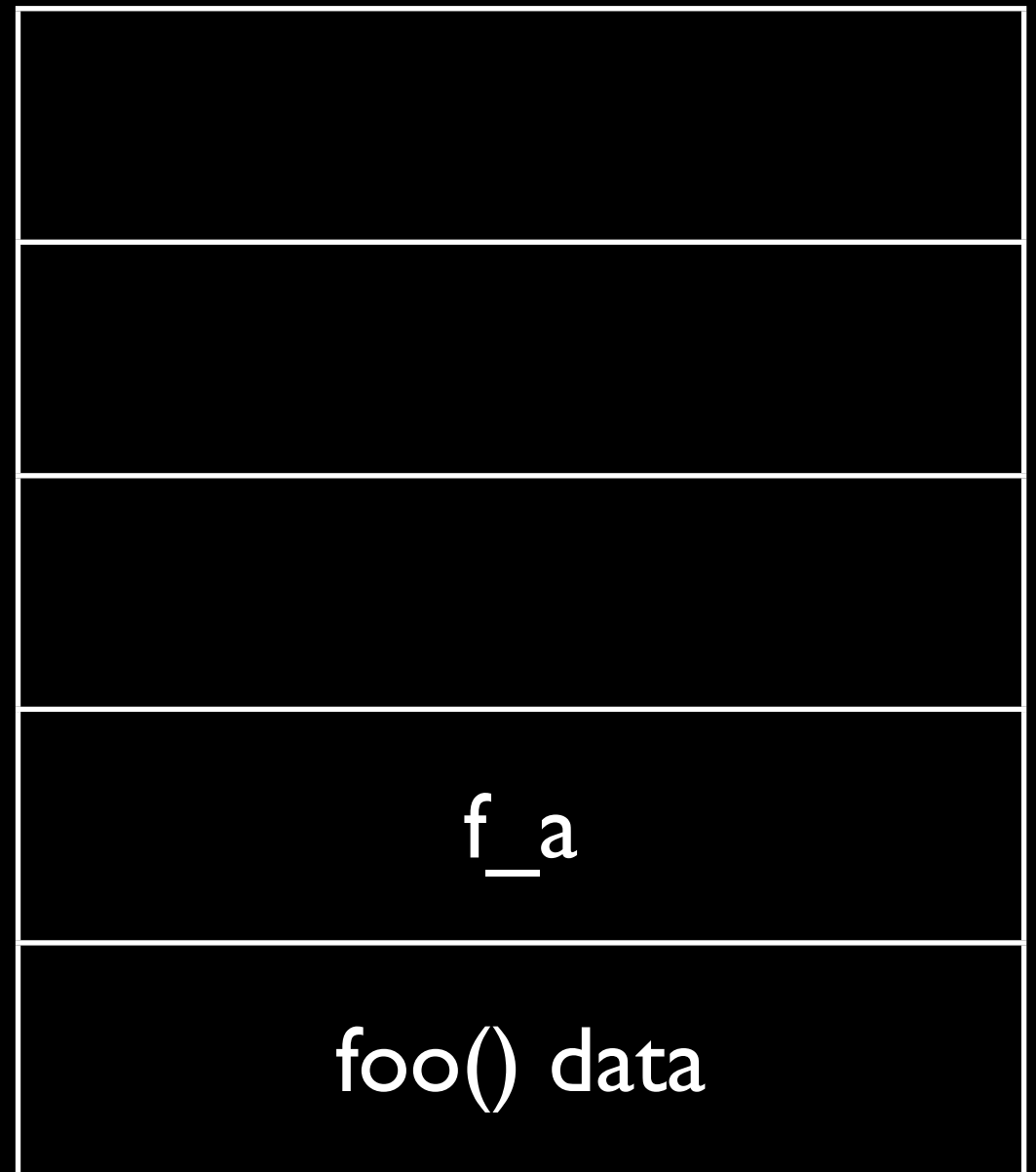
Case 2:

```
int bar() {  
    int b_a;  
}  
  
int foo() {  
    int f_a;  
    bar();  
    {  
        int f_b;  
    }  
    int f_c;  
}
```



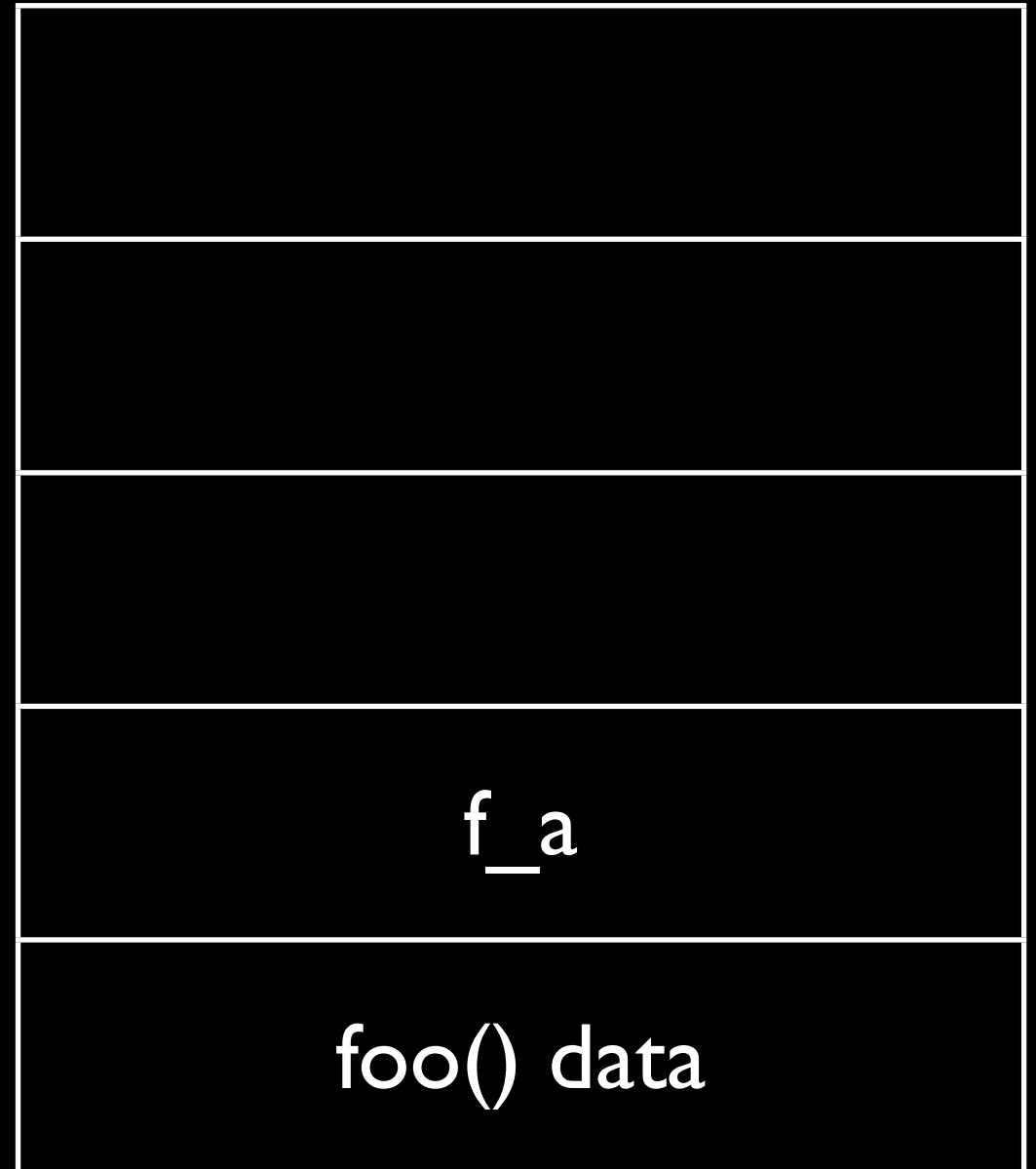
Case 2:

```
int bar() {  
    int b_a;  
}  
  
int foo() {  
    int f_a;  
    bar();  
    {  
        int f_b;  
    }  
    int f_c;  
}
```



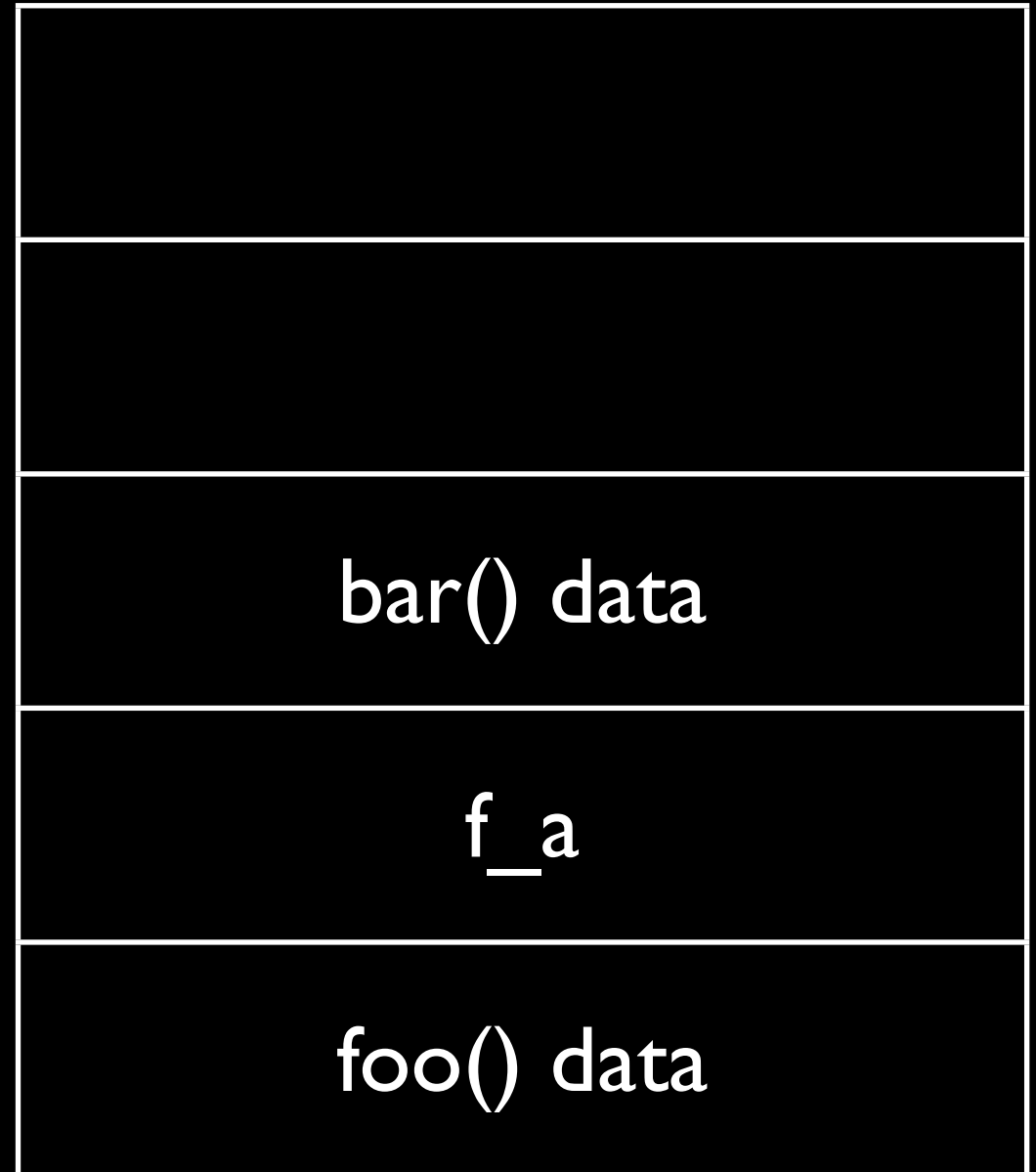
Case 2:

```
int bar() {  
    int b_a;  
}  
  
int foo() {  
    int f_a;  
    bar();  
    {  
        int f_b;  
    }  
    int f_c;  
}
```



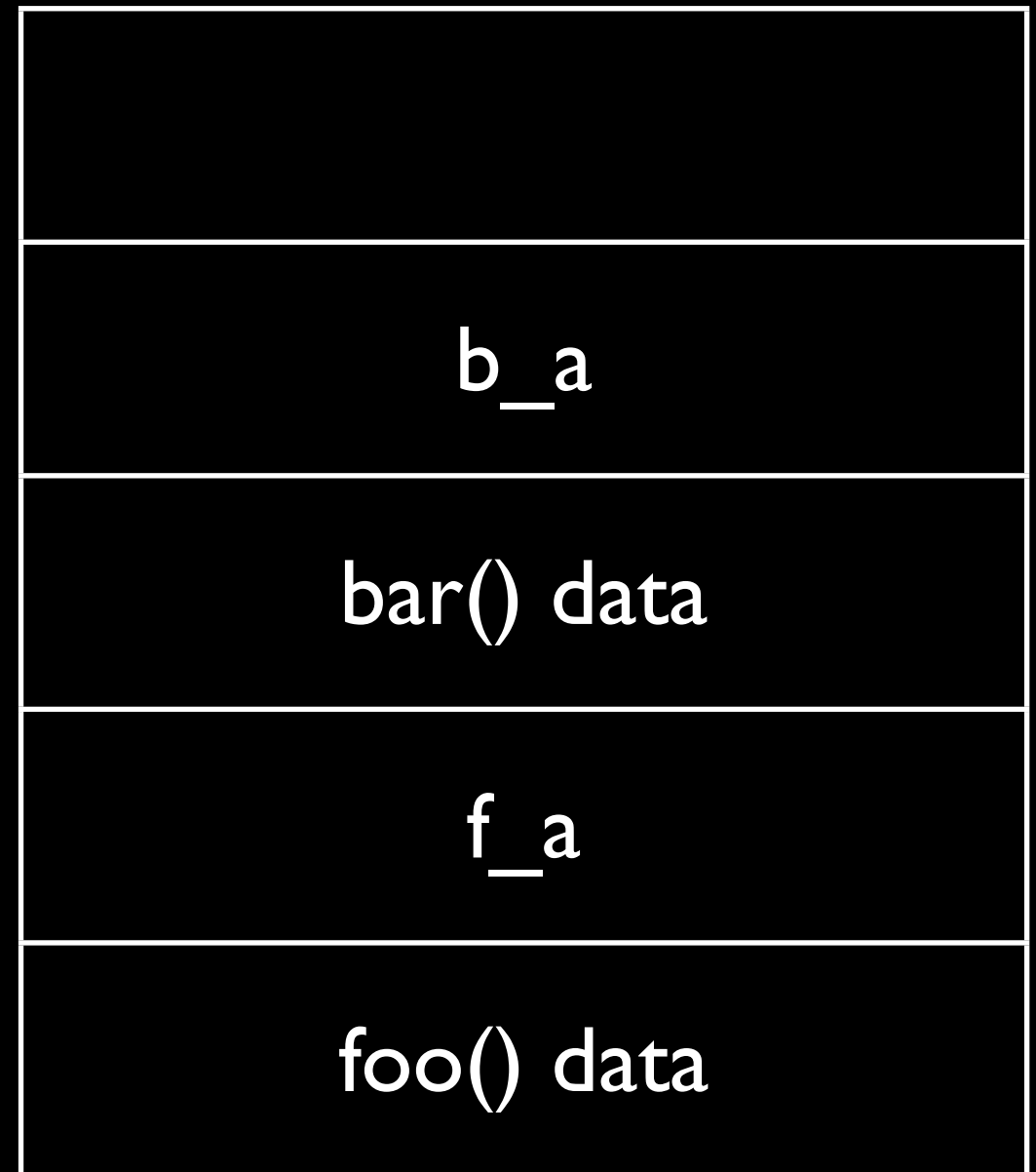
Case 2:

```
int bar() {  
    int b_a;  
}  
  
int foo() {  
    int f_a;  
    bar();  
    {  
        int f_b;  
    }  
    int f_c;  
}
```



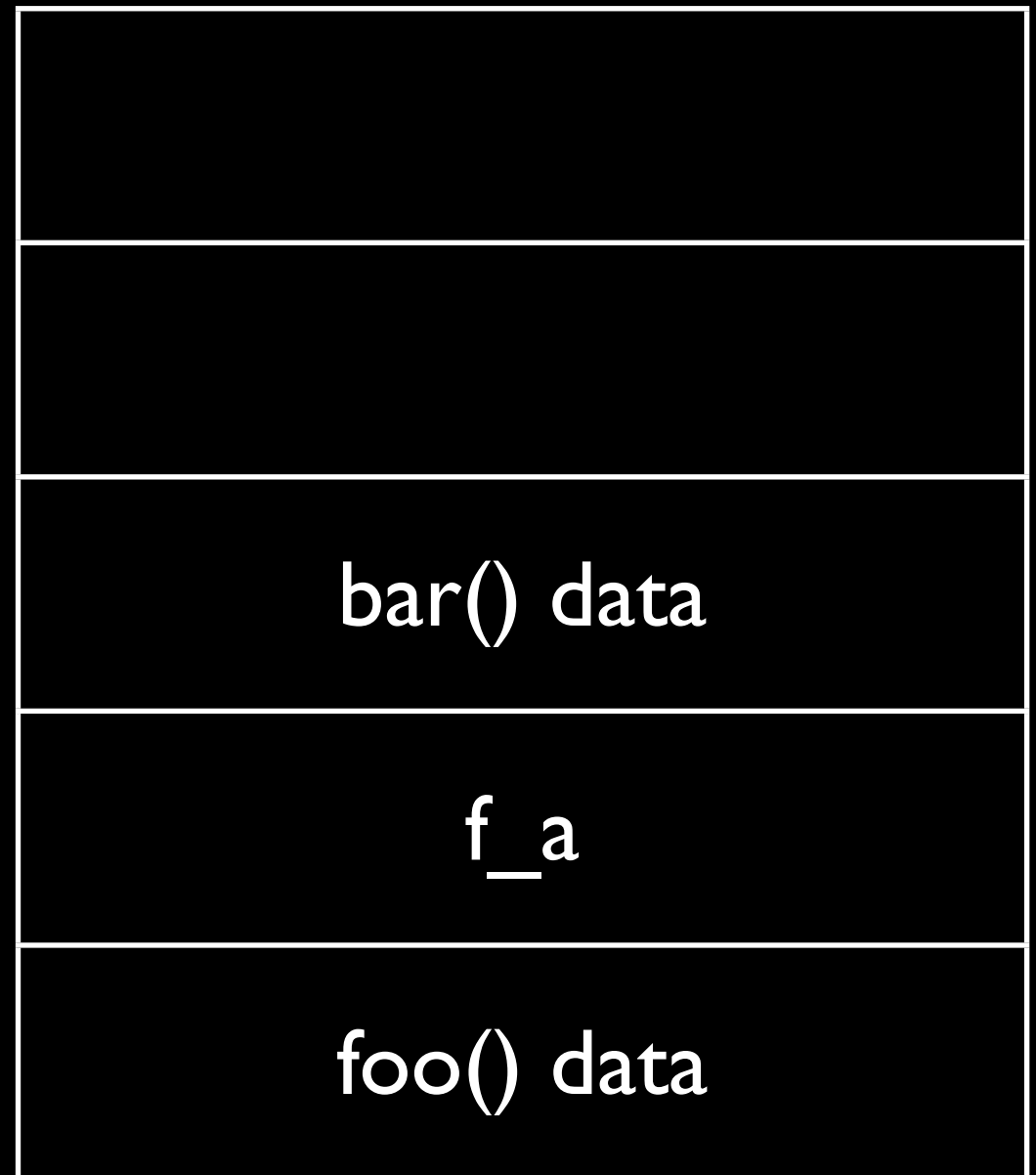
Case 2:

```
int bar() {  
    int b_a;  
}  
  
int foo() {  
    int f_a;  
    bar();  
    {  
        int f_b;  
    }  
    int f_c;  
}
```



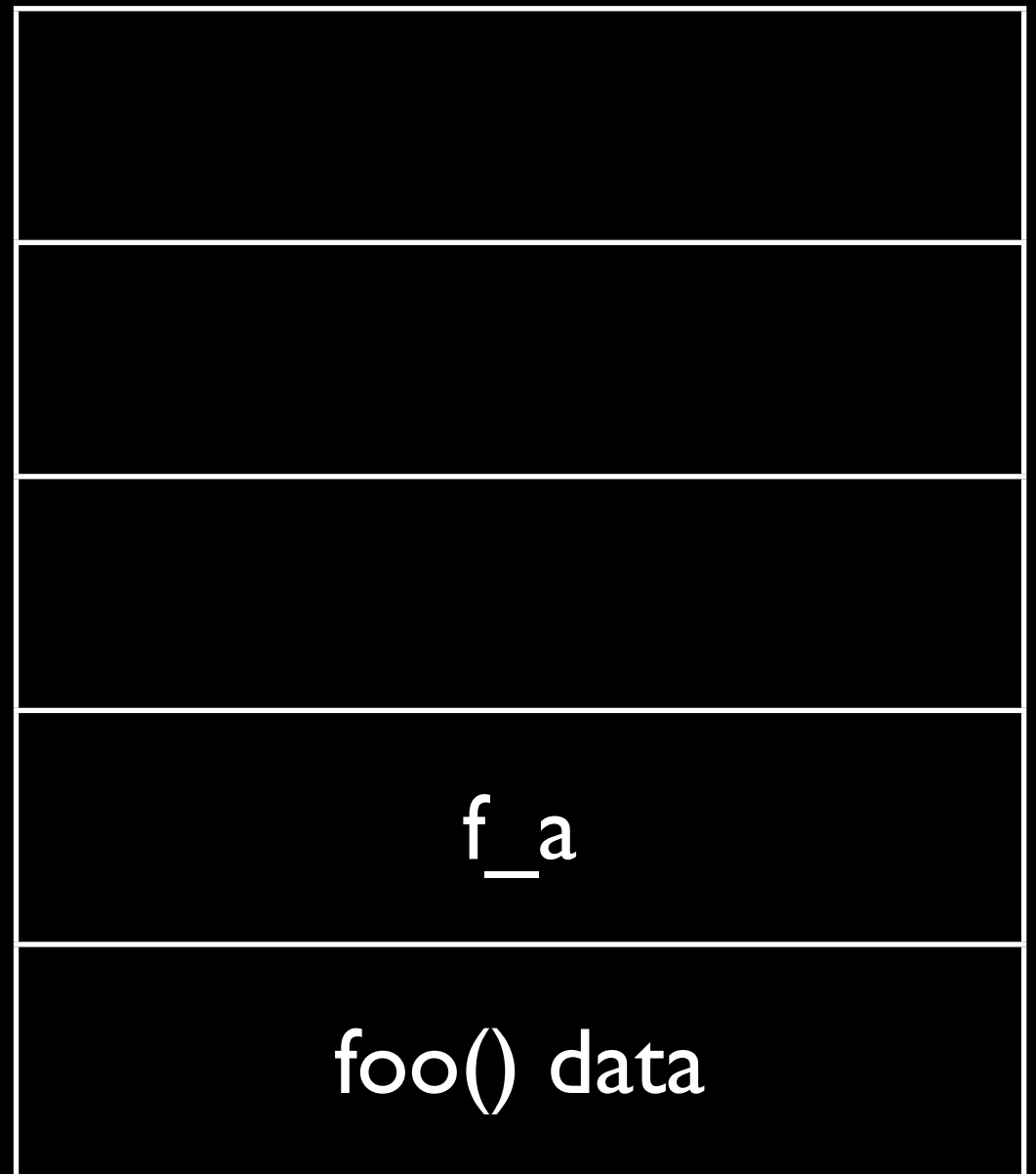
Case 2:

```
int bar() {  
    int b_a;  
}  
  
int foo() {  
    int f_a;  
    bar();  
    {  
        int f_b;  
    }  
    int f_c;  
}
```



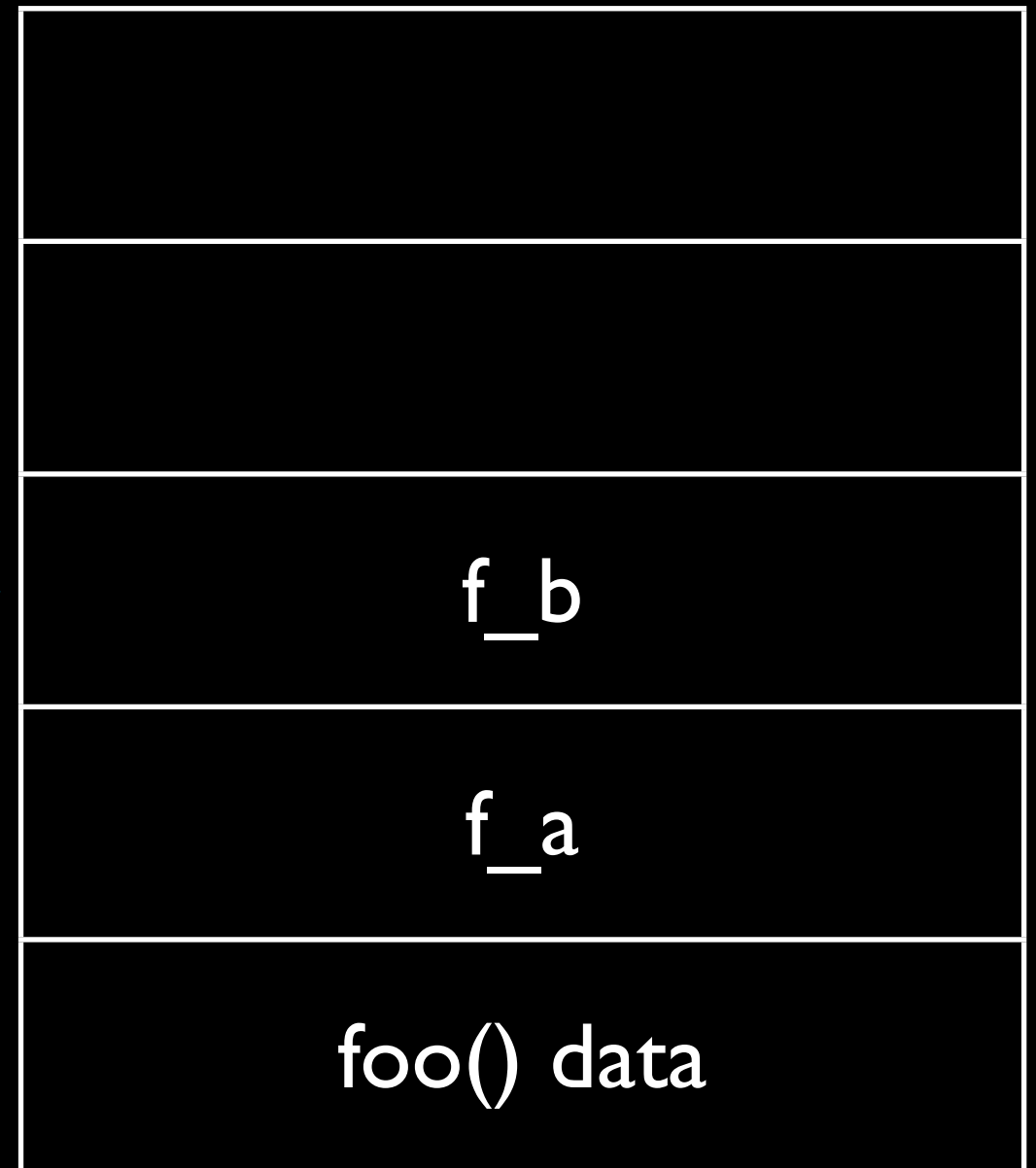
Case 2:

```
int bar() {  
    int b_a;  
}  
  
int foo() {  
    int f_a;  
    bar();  
    {  
        int f_b;  
    }  
    int f_c;  
}
```



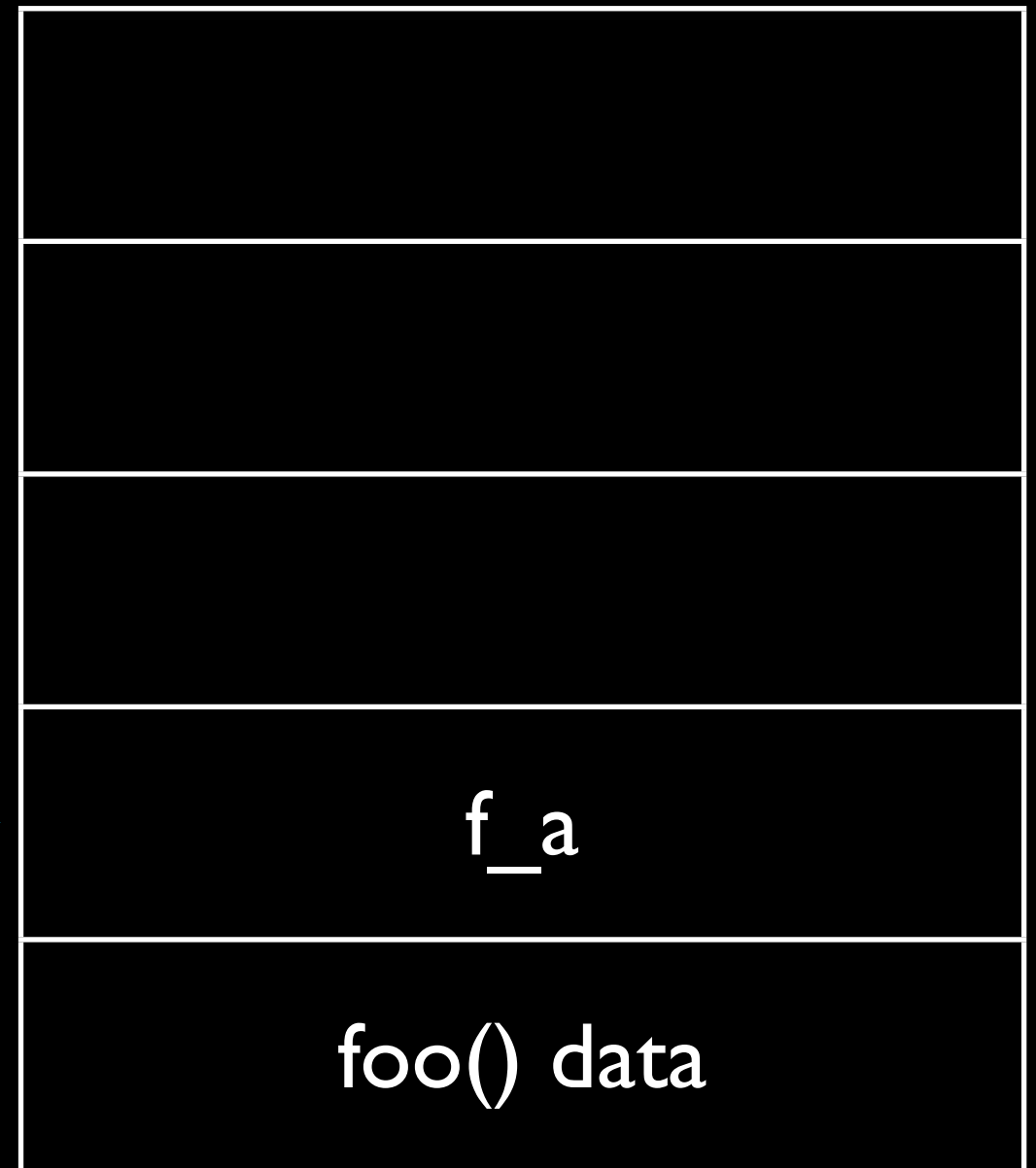
Case 2:

```
int bar() {  
    int b_a;  
}  
  
int foo() {  
    int f_a;  
    bar();  
    {  
        int f_b;  
    }  
    int f_c;  
}
```



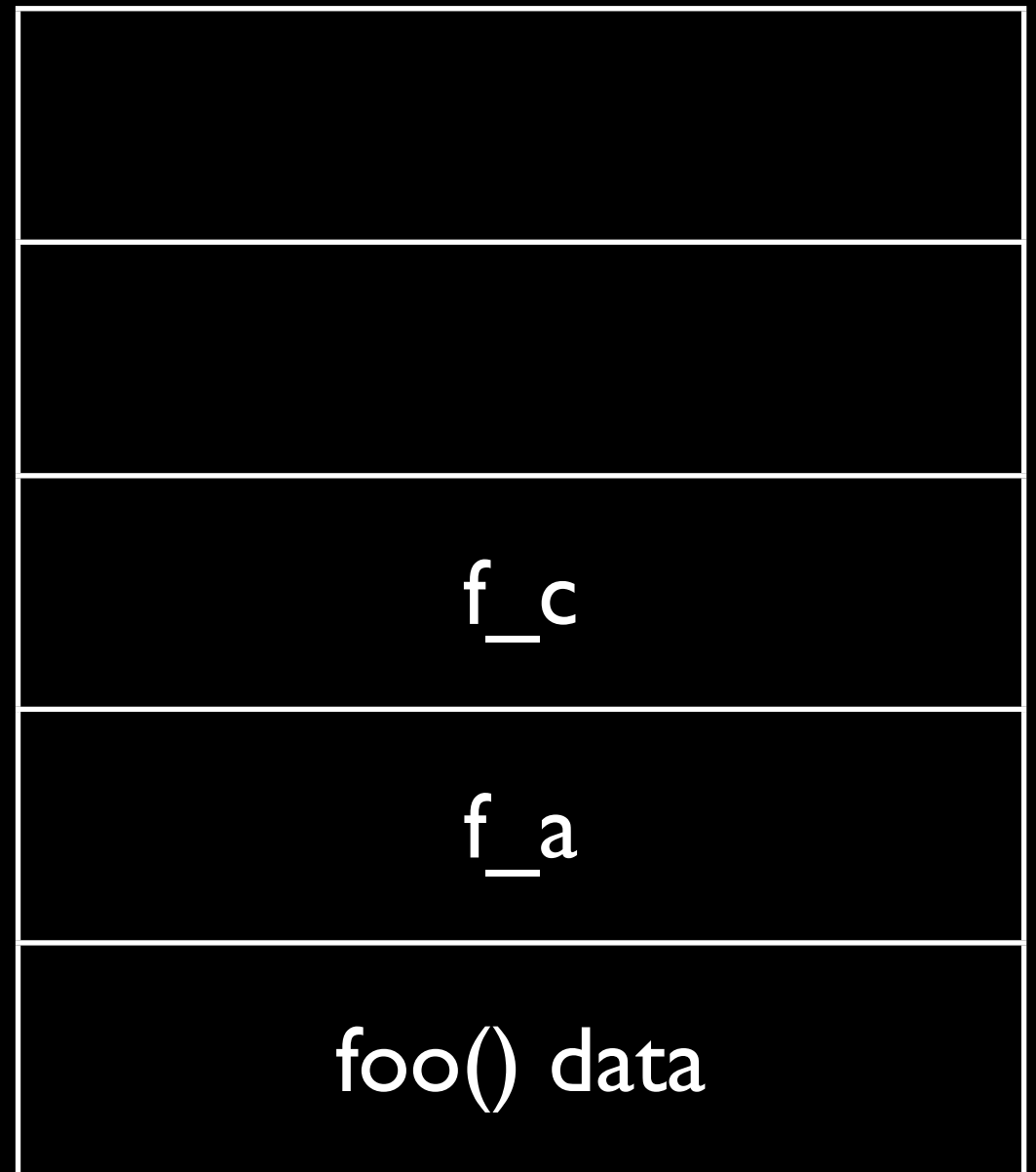
Case 2:

```
int bar() {  
    int b_a;  
}  
  
int foo() {  
    int f_a;  
    bar();  
    {  
        int f_b;  
    }  
    int f_c;  
}
```



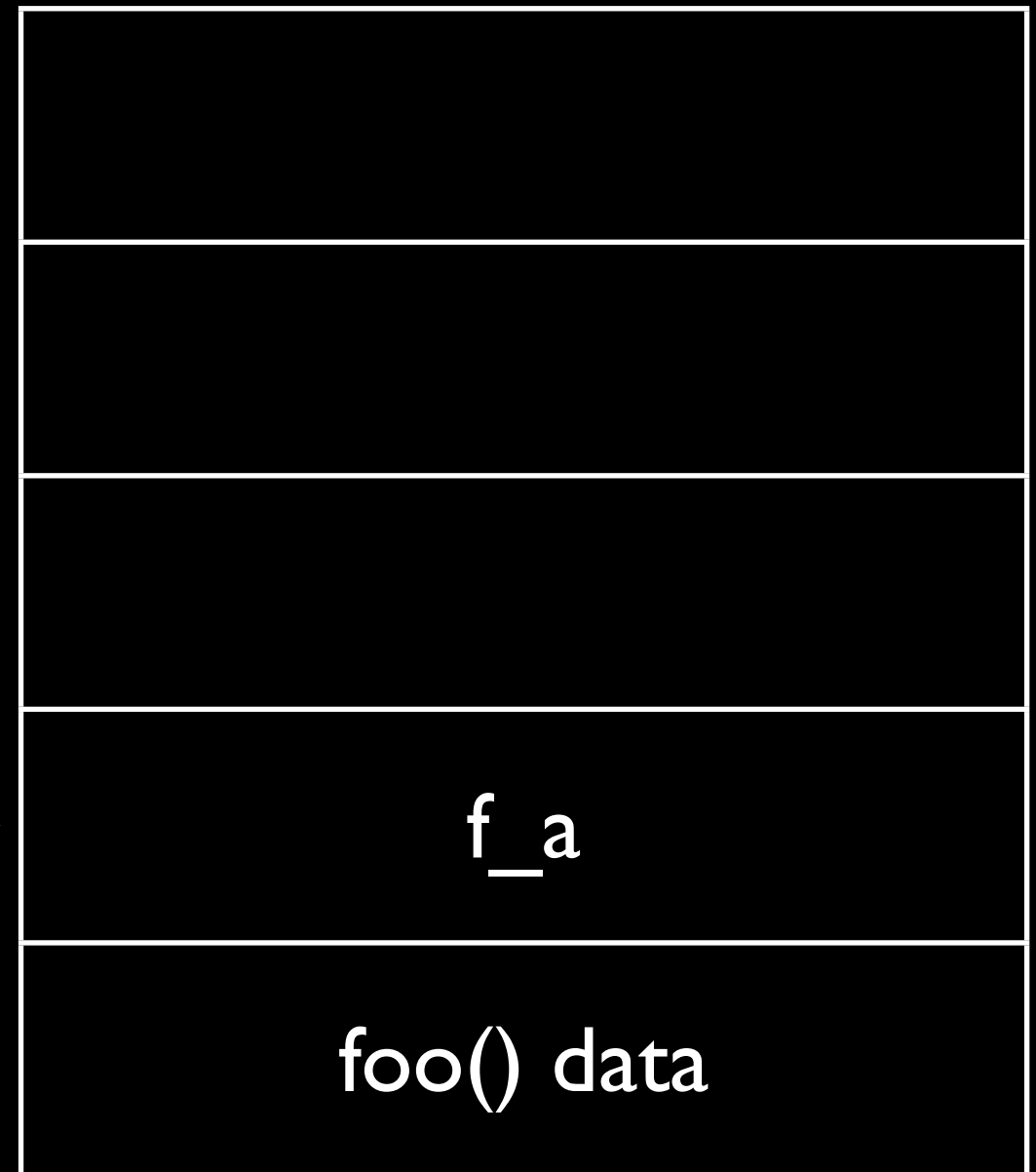
Case 2:

```
int bar() {  
    int b_a;  
}  
  
int foo() {  
    int f_a;  
    bar();  
    {  
        int f_b;  
    }  
    int f_c;  
}
```



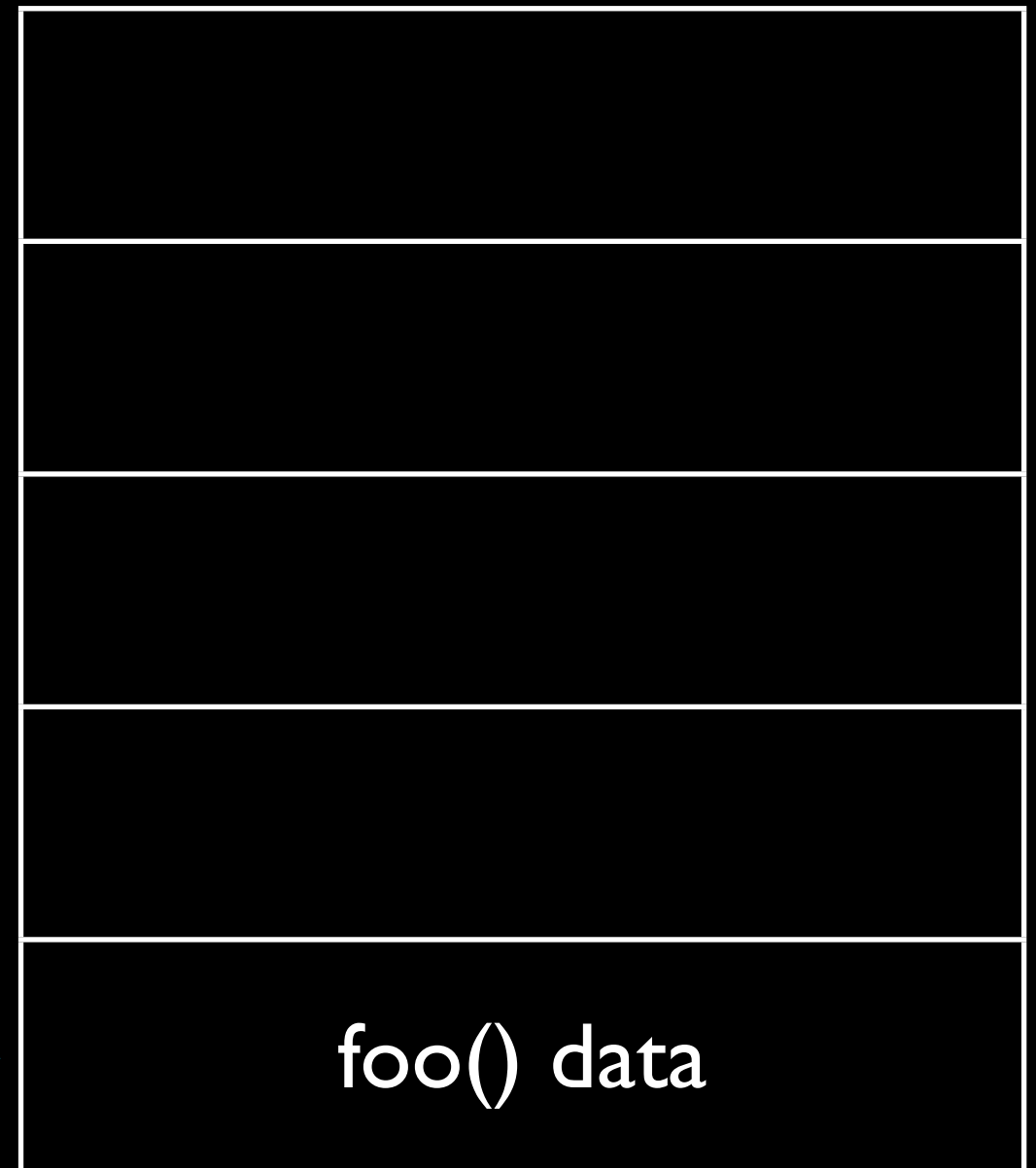
Case 2:

```
int bar() {  
    int b_a;  
}  
  
int foo() {  
    int f_a;  
    bar();  
    {  
        int f_b;  
    }  
    int f_c;  
}
```



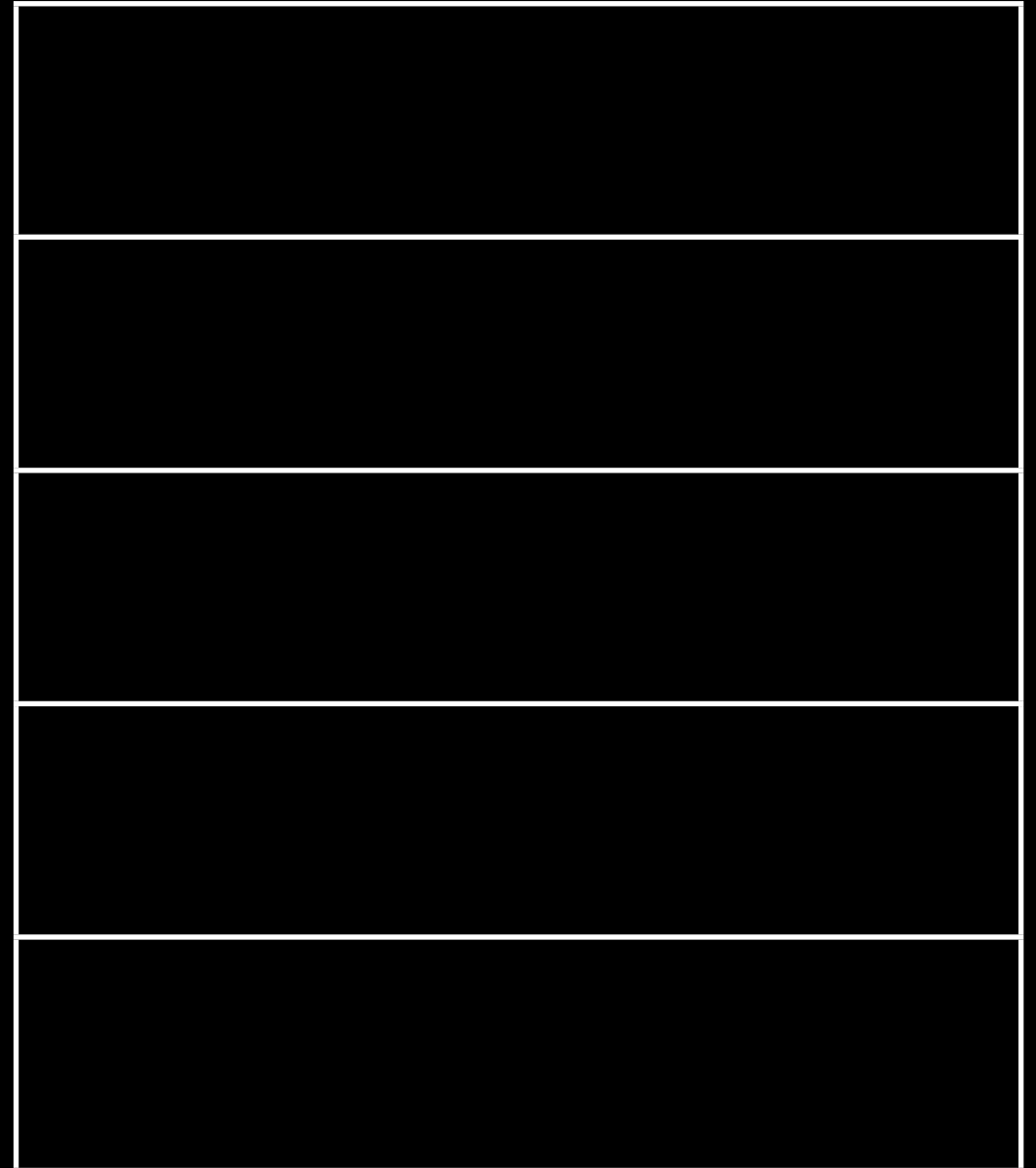
Case 2:

```
int bar() {  
    int b_a;  
}  
  
int foo() {  
    int f_a;  
    bar();  
    {  
        int f_b;  
    }  
    int f_c;  
}
```



Case 2:

```
int bar() {  
    int b_a;  
}  
  
int foo() {  
    int f_a;  
    bar();  
    {  
        int f_b;  
    }  
    int f_c;  
}
```



And this is why it's called *stack* memory.

sizeof

- Returns the size in bytes of the data type.
- Useful when dealing with low-level stuff.

```
cout << "sizeof(int)" << sizeof(int)
      << "\nsizeof(char)" << sizeof(char)
      << "\nsizeof(int*)" << sizeof(int*)
      << "\nsizeof(double)" << sizeof(double)
      << "\nsizeof(float)" << sizeof(float);
```

Checkpoint 2.10

Compare the output of the `sizeof` in the 3 kinds of array and explain the difference (note: `sizeof(int) == 4`).

```
int a[5];
int *b = new int[5];
vector<int> c;
for ( int i = 0; i < 5; ++i )
    c.push_back(i);

cout << "a: " << sizeof(a)
     << "\nb: " << sizeof(b)
     << "\nc: " << sizeof(c);
```

Checkpoint 2.11

What does the second line evaluate to?

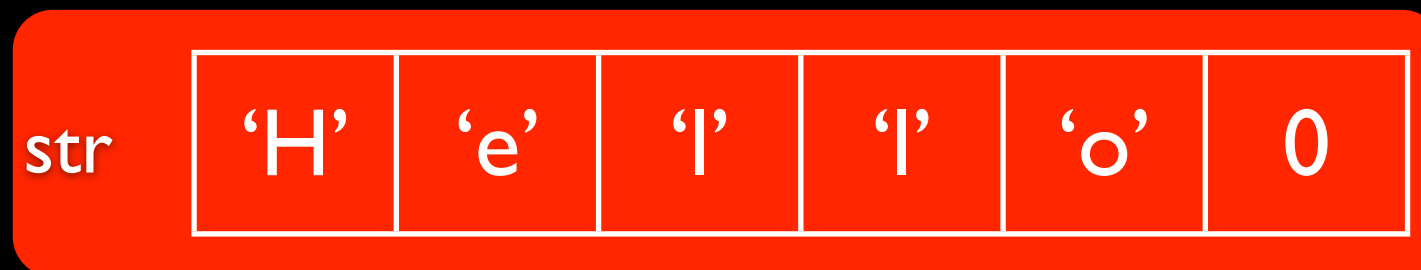
```
int a[10];  
sizeof(a)/sizeof(a[0]);
```

Will it also work for built-in dynamic arrays?

Strings

- A string is essentially *an array of characters* terminated by a “null character” (a zero).
- This kind of representation is often called a “C-string”.
- The total size of the storage is 1 more than the length of the string.

```
char str[] = "Hello";
```



String length is 5 but actual array size is 6.

Strings

- Since all C-strings are terminated by a null character, the length of the string can be derived from the data itself.

```
int stringLength(char *p) {  
    int len = 0;  
    while ( p[len] != 0 )  
        ++len;  
    return len;  
}
```

Null Character

- Also called “end-of-string sentinel” or “null terminator”
- The integer value is 0
- The escape character is '\0'
- They can be used interchangeably.
- The string literal automatically appends the null terminator.

```
int stringLength(char *p) {  
    int len = 0;  
    while ( p[len] != '\0' )  
        ++len;  
    return len;  
}
```

String Concatenation

```
char* stringJoin(char *a, char *b) {  
    int lenA = stringLength(a), lenB = stringLength(b);  
    char *ret = new char[lenA + lenB + 1];  
  
    for( int i = 0; a[i] != 0; ++i)  
        ret[i] = a[i];  
    for ( int i = 0; b[i] != 0; ++i )  
        ret[i + lenA] = b[i];  
  
    ret[lenA + lenB] = 0;  
    return ret;  
}
```

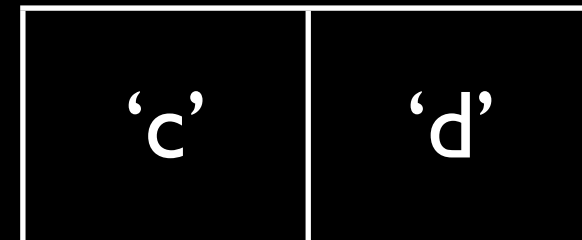
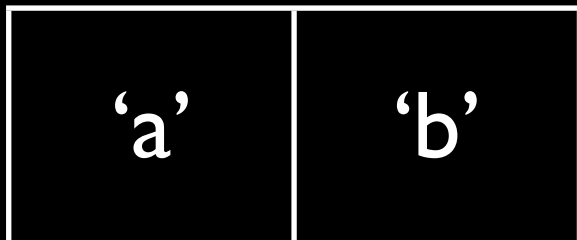
String Concatenation

```
char* stringJoin(char *a, char *b) {  
    int lenA = stringLength(a), lenB = stringLength(b);  
    char *ret = new char[lenA + lenB + 1];  
  
    for( int i = 0; a[i] != 0; ++i)  
        ret[i] = a[i];  
    for ( int i = 0; b[i] != 0; ++i )  
        ret[i + lenA] = b[i];  
  
    ret[lenA + lenB] = 0;  
    return ret;  
}
```

```
char *c = stringJoin("ab", "cd");
```


String Concatenation

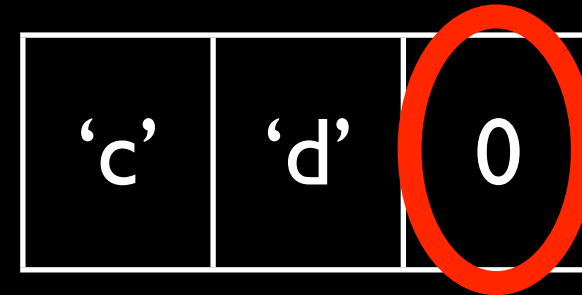
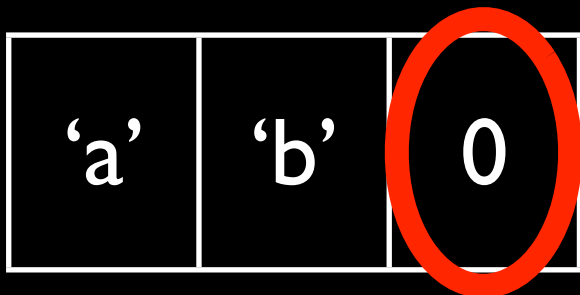
```
char* stringJoin(char *a, char *b) {  
    int lenA = stringLength(a), lenB = stringLength(b);  
    char *ret = new char[lenA + lenB + 1];  
  
    for( int i = 0; a[i] != 0; ++i)  
        ret[i] = a[i];  
    for ( int i = 0; b[i] != 0; ++i )  
        ret[i + lenA] = b[i];  
  
    ret[lenA + lenB] = 0;  
    return ret;  
}
```



Something seems missing...

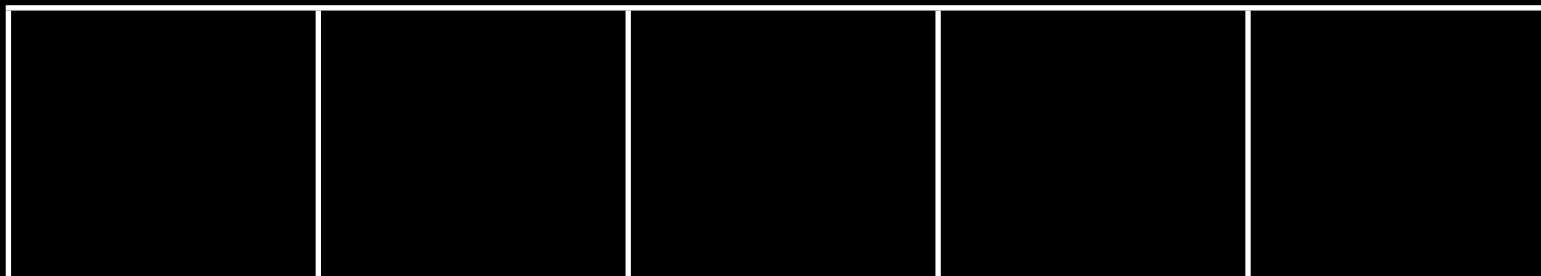
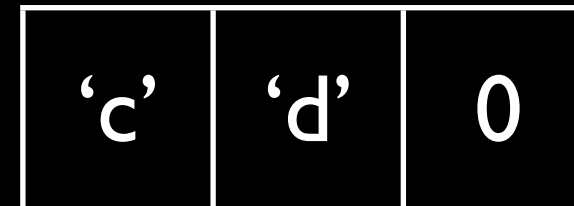
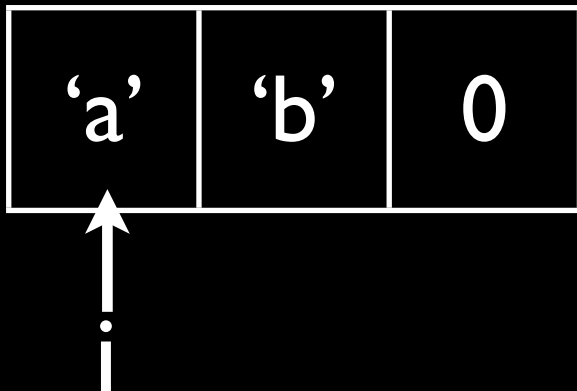
String Concatenation

```
char* stringJoin(char *a, char *b) {  
    int lenA = stringLength(a), lenB = stringLength(b);  
    char *ret = new char[lenA + lenB + 1];  
  
    for( int i = 0; a[i] != 0; ++i)  
        ret[i] = a[i];  
    for ( int i = 0; b[i] != 0; ++i )  
        ret[i + lenA] = b[i];  
  
    ret[lenA + lenB] = 0;  
    return ret;  
}
```



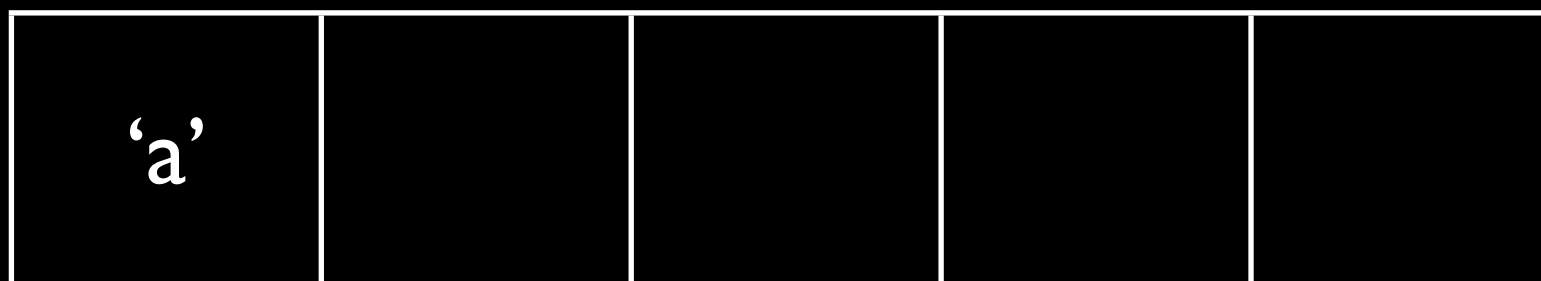
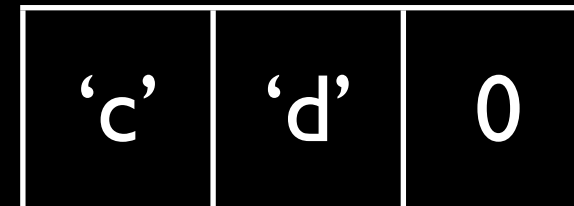
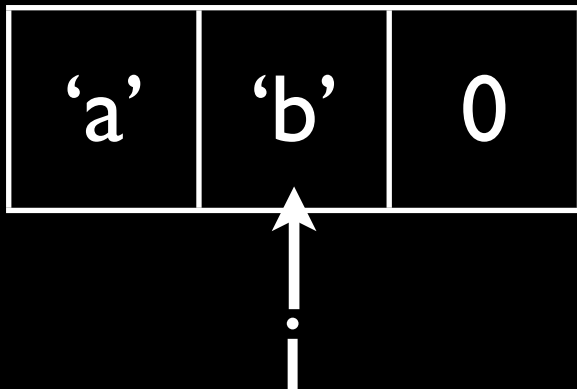
String Concatenation

```
char* stringJoin(char *a, char *b) {  
    int lenA = strlen(a), lenB = strlen(b);  
    char *ret = new char[lenA + lenB + 1];  
  
    for( int i = 0; a[i] != 0; ++i)  
        ret[i] = a[i];  
    for ( int i = 0; b[i] != 0; ++i )  
        ret[i + lenA] = b[i];  
  
    ret[lenA + lenB] = 0;  
    return ret;  
}
```



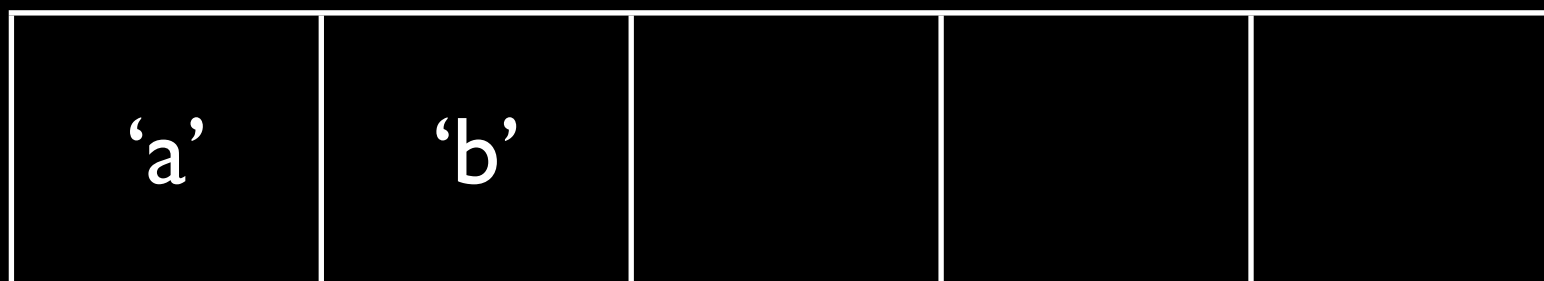
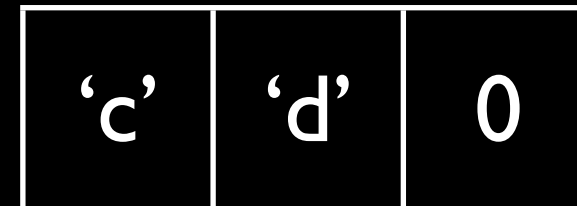
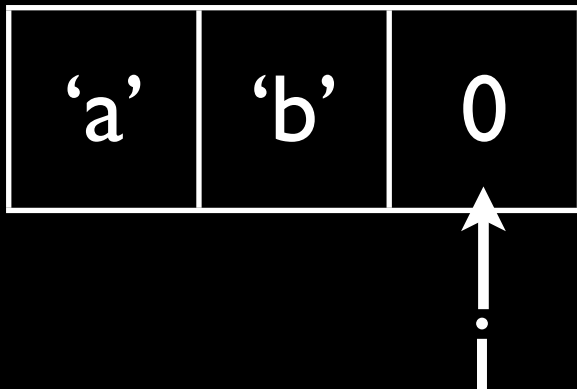
String Concatenation

```
char* stringJoin(char *a, char *b) {  
    int lenA = strlen(a), lenB = strlen(b);  
    char *ret = new char[lenA + lenB + 1];  
  
    for( int i = 0; a[i] != 0; ++i)  
        ret[i] = a[i];  
    for ( int i = 0; b[i] != 0; ++i )  
        ret[i + lenA] = b[i];  
  
    ret[lenA + lenB] = 0;  
    return ret;  
}
```



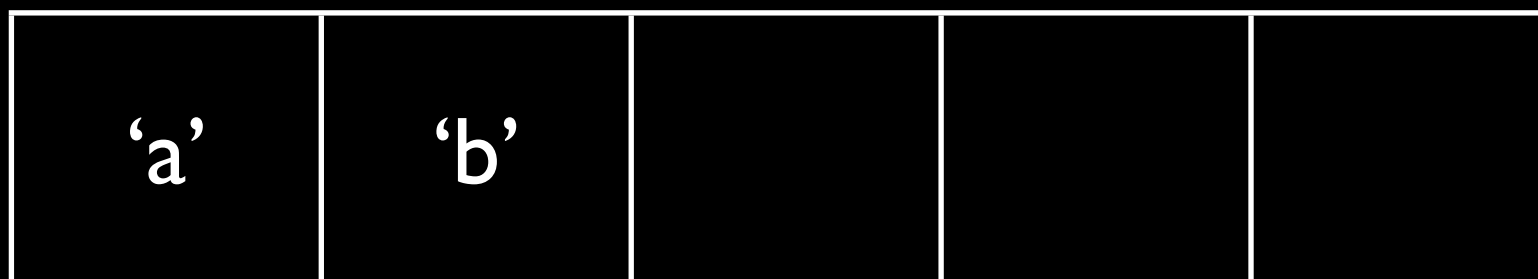
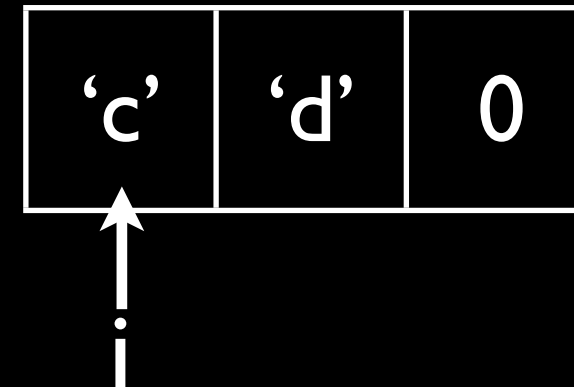
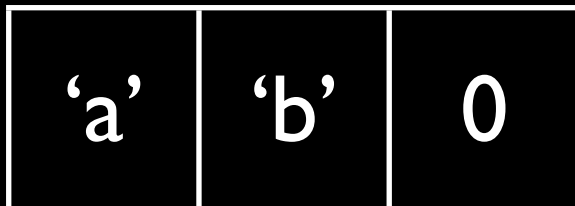
String Concatenation

```
char* stringJoin(char *a, char *b) {  
    int lenA = strlen(a), lenB = strlen(b);  
    char *ret = new char[lenA + lenB + 1];  
  
    for( int i = 0; a[i] != 0; ++i)  
        ret[i] = a[i];  
    for ( int i = 0; b[i] != 0; ++i )  
        ret[i + lenA] = b[i];  
  
    ret[lenA + lenB] = 0;  
    return ret;  
}
```



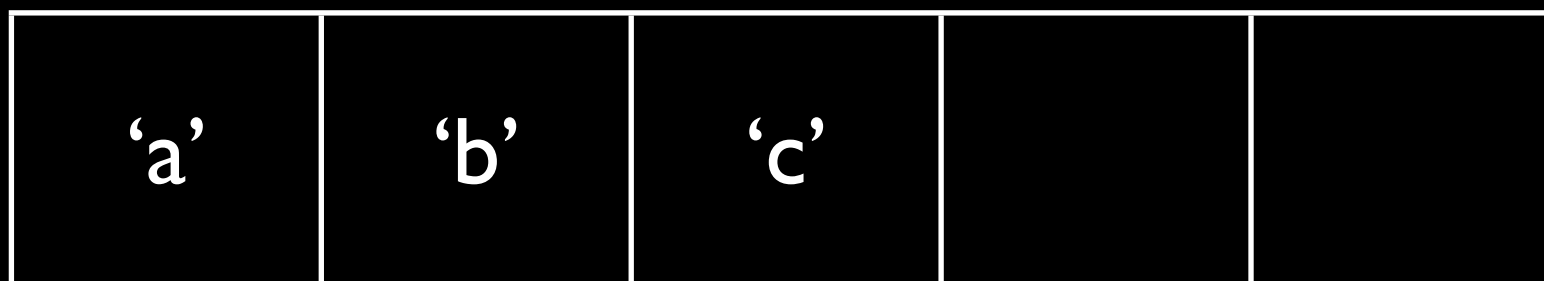
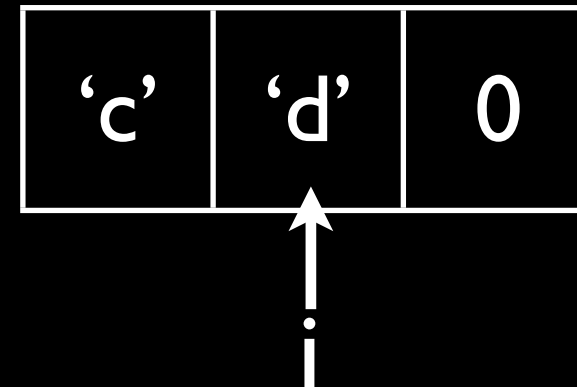
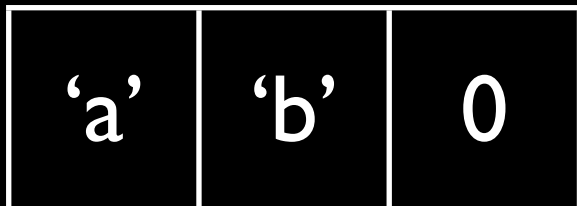
String Concatenation

```
char* stringJoin(char *a, char *b) {  
    int lenA = stringLength(a), lenB = stringLength(b);  
    char *ret = new char[lenA + lenB + 1];  
  
    for( int i = 0; a[i] != 0; ++i)  
        ret[i] = a[i];  
    for ( int i = 0; b[i] != 0; ++i )  
        ret[i + lenA] = b[i];  
  
    ret[lenA + lenB] = 0;  
    return ret;  
}
```



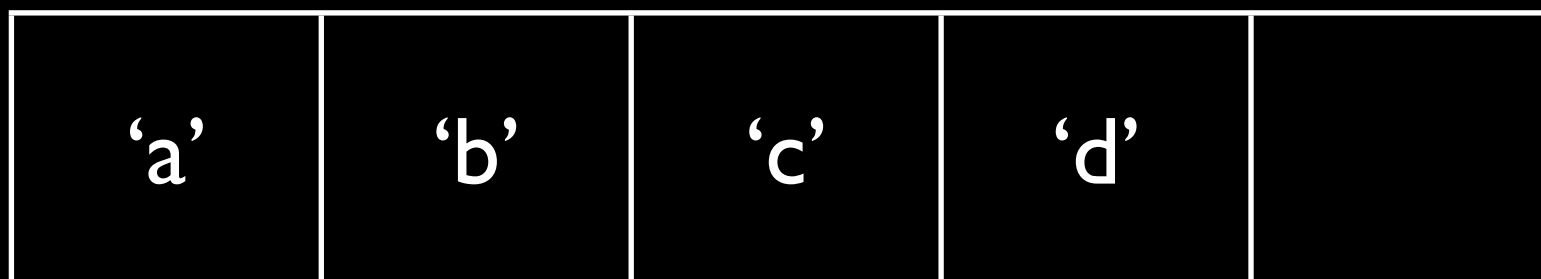
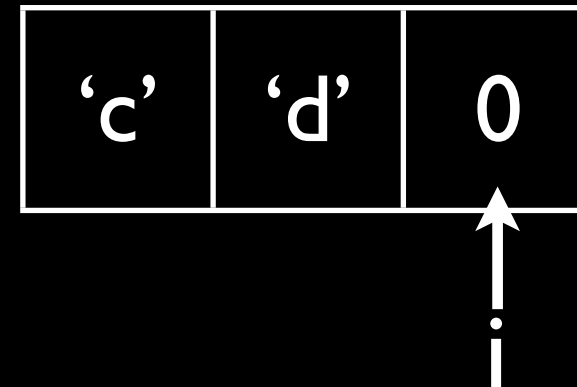
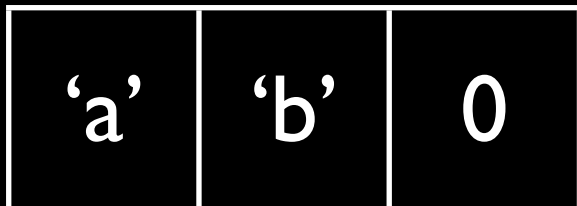
String Concatenation

```
char* stringJoin(char *a, char *b) {  
    int lenA = stringLength(a), lenB = stringLength(b);  
    char *ret = new char[lenA + lenB + 1];  
  
    for( int i = 0; a[i] != 0; ++i)  
        ret[i] = a[i];  
    for ( int i = 0; b[i] != 0; ++i )  
        ret[i + lenA] = b[i];  
  
    ret[lenA + lenB] = 0;  
    return ret;  
}
```



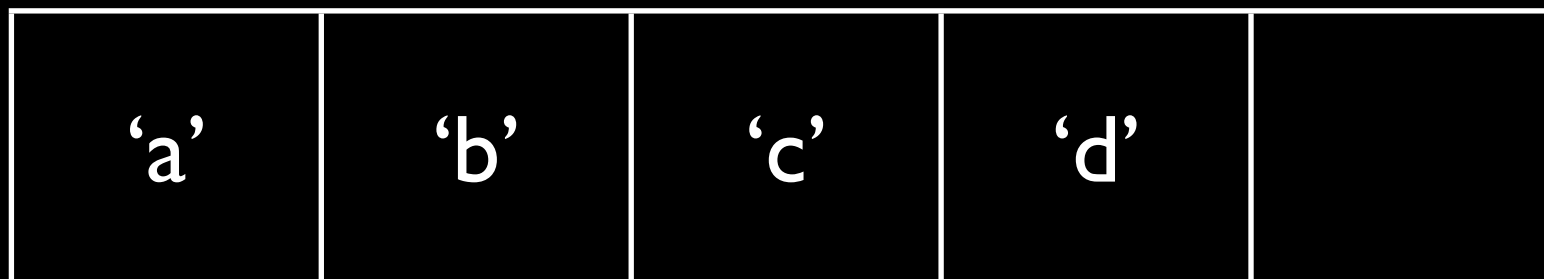
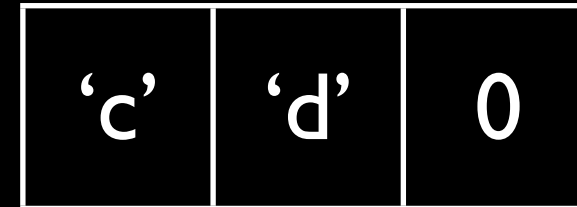
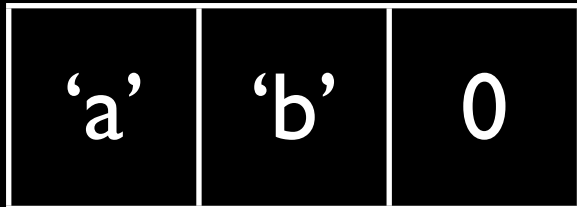
String Concatenation

```
char* stringJoin(char *a, char *b) {  
    int lenA = strlen(a), lenB = strlen(b);  
    char *ret = new char[lenA + lenB + 1];  
  
    for( int i = 0; a[i] != 0; ++i)  
        ret[i] = a[i];  
    for ( int i = 0; b[i] != 0; ++i )  
        ret[i + lenA] = b[i];  
  
    ret[lenA + lenB] = 0;  
    return ret;  
}
```



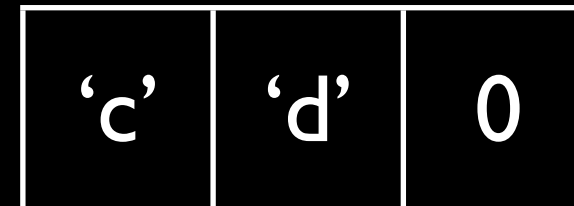
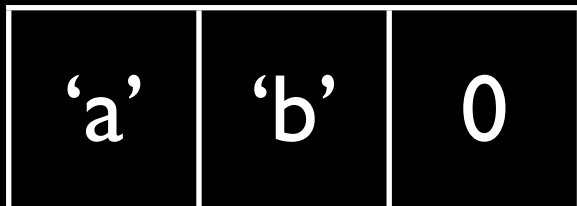
String Concatenation

```
char* stringJoin(char *a, char *b) {  
    int lenA = strlen(a), lenB = strlen(b);  
    char *ret = new char[lenA + lenB + 1];  
  
    for( int i = 0; a[i] != 0; ++i)  
        ret[i] = a[i];  
    for ( int i = 0; b[i] != 0; ++i )  
        ret[i + lenA] = b[i];  
  
    ret[lenA + lenB] = 0;  
    return ret;  
}
```

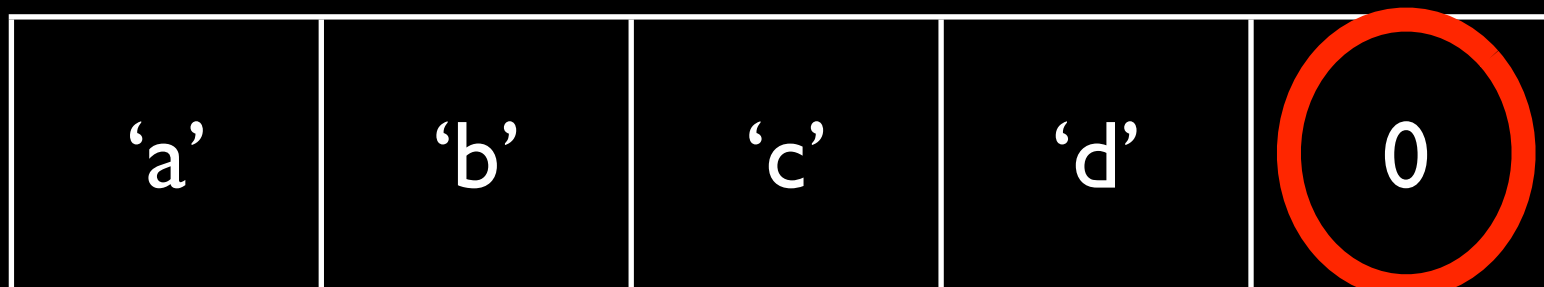


String Concatenation

```
char* stringJoin(char *a, char *b) {  
    int lenA = strlen(a), lenB = strlen(b);  
    char *ret = new char[lenA + lenB + 1];  
  
    for( int i = 0; a[i] != 0; ++i)  
        ret[i] = a[i];  
    for ( int i = 0; b[i] != 0; ++i )  
        ret[i + lenA] = b[i];  
  
    ret[lenA + lenB] = 0;  
    return ret;  
}
```



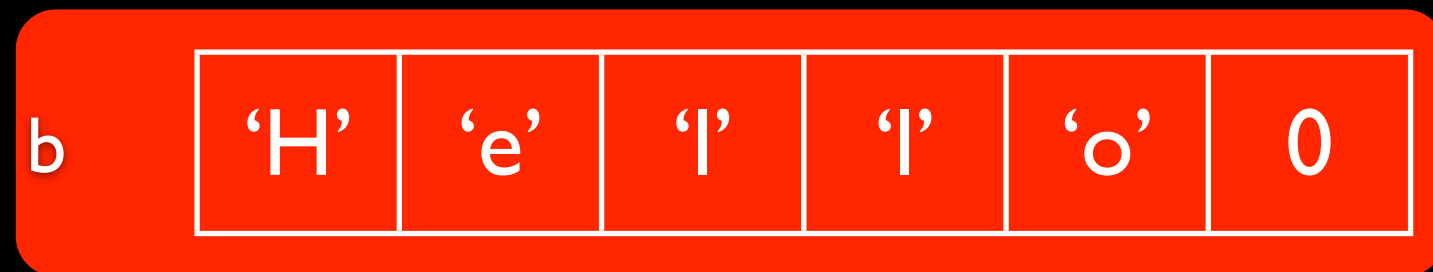
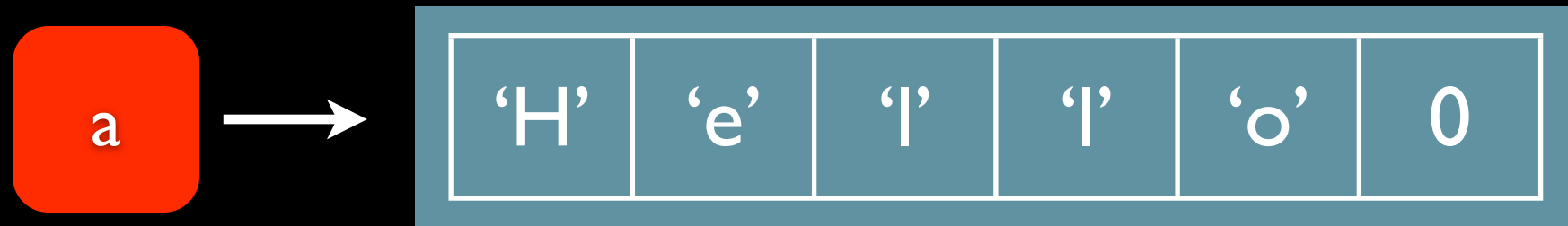
It is **very** important to terminate with a null!



Checkpoint 2.12

What's the difference between the two?

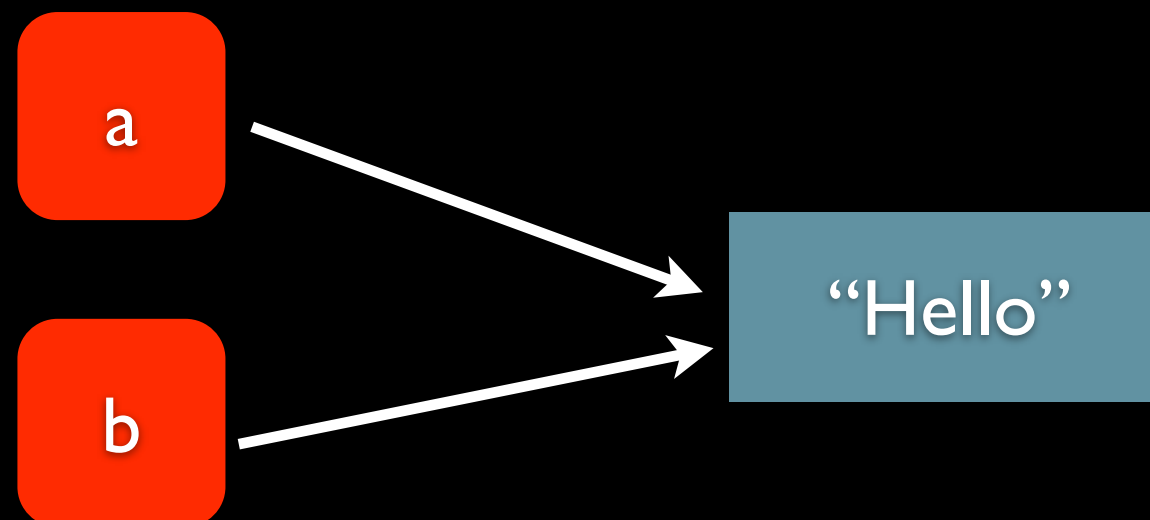
```
char *a = "Hello";  
char b[] = "Hello";
```



Checkpoint 2.13

What happens in the end? Is this ok?

```
char *a = "Hello";  
char *b = "Bye";  
b = a;
```



C-String helpers

- Checkout `<cstring>` for helper functions.
- Some notable functions: `strlen`, `strcat`, `strtok`, `sprintf`, `sscanf`

C++ Strings

- More often than not, C++ classes wrap string functionality.
- `std::string` is the usual but some frameworks provide their own (e.g. `QString`, `WtString`, etc.)

```
using namespace std;  
string a = "Hello";  
string b = "World";  
string c = a + b;
```

C++ Strings

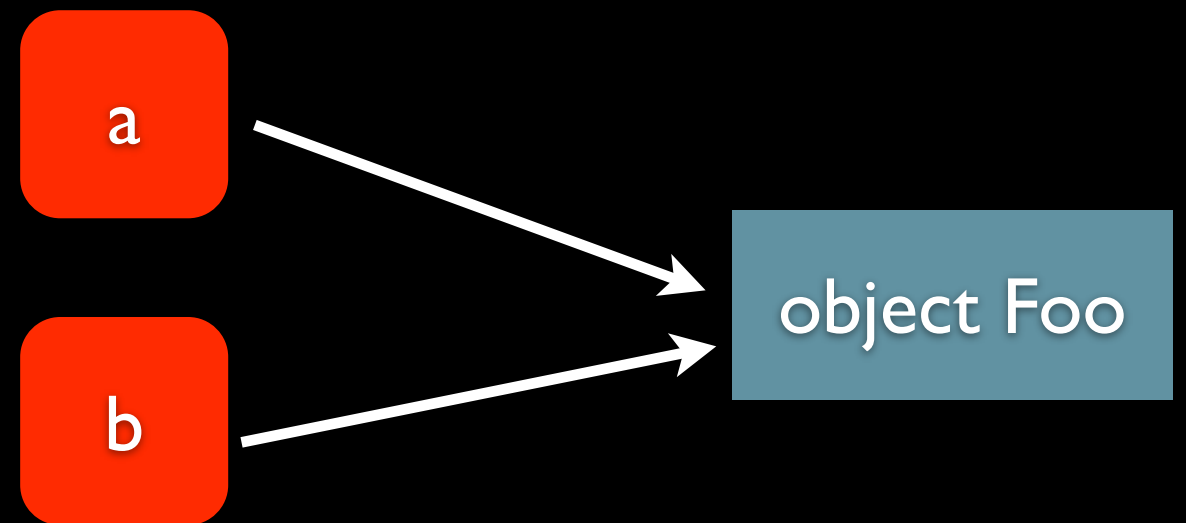
- Individual characters may also be accessed using the `[]` operator.
- Like `std::vector<>`, pass them as (const) references.
- Has the `.length()` and `.size()` property (they return the same value)
- `.c_str()` returns the `const char *` equivalent of the string (valid until the string changes).

BREAKT13M

Dynamic Allocation

Java

```
Foo a = new Foo();  
Foo b = a;
```



C++

```
Foo a;  
Foo *b = &a;
```



Dynamic Allocation

```
Foo* fooFactory() {  
    Foo *foo = //allocate new foo instance;  
    return foo;  
}
```

```
Foo* fooFactory() {  
    Foo foo;  
    return &foo;  
}
```



Do not want.

```
Foo* fooFactory() {  
    static Foo foo;  
    return &foo;  
}
```



Not sure if want.
(depends)

Dynamic Allocation

```
Foo* fooFactory() {  
    Foo *foo = new Foo();  
    return &foo;  
}
```

`new ClassName(parameters)` returns a *pointer* of type `ClassName`.

Allocates memory from the *heap* and not from the stack.

Dynamic Allocation

- C++ is not a Garbage collected language.
- Dynamically allocated objects have to be manually `delete`-d;
- Failure to do so results in a *memory leak*.
- The OS usually cleans up all resources when a program exits.

```
Foo *foo = fooFactory();  
//use foo...  
delete foo;
```

Dynamic Allocation

Note that there are two forms of delete.

Use the appropriate one.

Failure to do so will result in *undefined behavior*.

Foo *a = new Foo(); → delete a;

Foo *b = new Foo[ARR_SIZE]; → delete [] b;

No need to specify the size of the array for the array delete form.

Dynamic Allocation.

- Do not delete the same address twice.
- Deleting a **NULL** pointer is safe.
- A pointer is usually set to **NULL** after deleting to prevent double-deletion.

```
Foo *a = new Foo();  
delete a;  
a = 0;
```

Dynamic Allocation

- Java-style array allocation can be done.
- Allocate pointers instead of the actual object.
- Just don't forget to delete them after.

```
std::vector<Foo*> a;  
a.push_back(new Foo());
```

```
Foo* b[10];  
b[0] = new Foo();
```

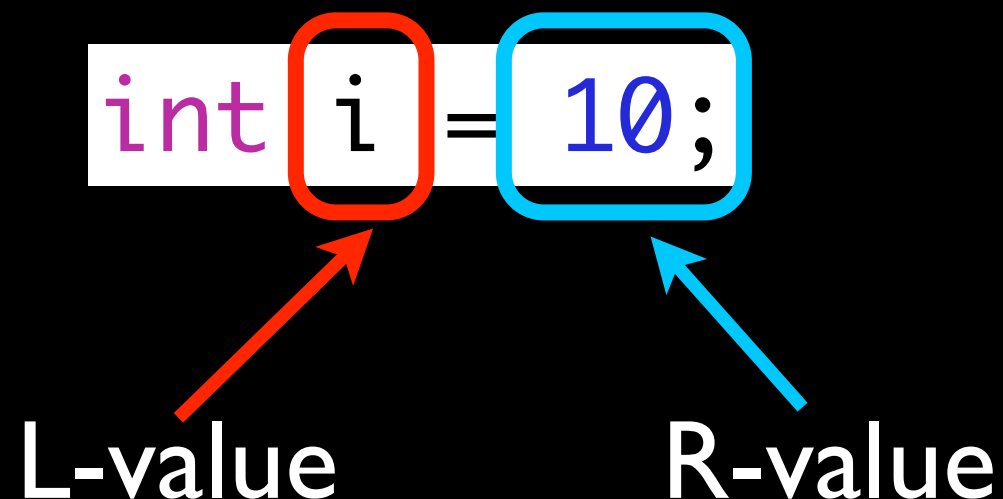
Checkpoint 2.14

What's possibly wrong with the following code?

```
void initArray(vector<Foo*> &v, const int n) {  
    for ( int i = 0; i < n; ++i )  
        v.push_back(new Foo());  
}  
  
void doFoo() {  
    vector<Foo*> v;  
    initArray(v, 10);  
    v.clear();  
}
```


L-values and R-values

- R-values: values that can only be placed on the right.
- L-values: values that can be placed on the left.
- Constants are considered L-values.



Examples 0

Red: L-value

Blue: R-value

```
int i = 10;  
int a = 20;  
a = i;  
i = 20;  
20 = 10; ← Not allowed!  
const int a = 20;
```

Examples I

Red: L-value

Blue: R-value

```
Foo a[10];  
a[0] = 0;  
Foo* b;  
b = a;  
void doBar() {}  
void (*funPtr)();  
funPtr = doBar;
```

Examples 2

Red: L-value

Blue: R-value

```
int a = 1 + 2;  
int b = 2 * 2;  
int c = a + b;  
int bar() { return 0; }  
int foo() {  
    int a = 10;  
    return a;  
}  
a = bar() + foo();
```

Examples 3

Red: L-value

Blue: R-value

```
int a = 0;  
int b = 2;  
int& getA() { return a; }  
int getB() { return b; }  
getA() = 1;  
int c = getA() + getB();
```

L-values and R-values

- Only L-values have addresses.
- Non-const references can only capture L-values.
- Const references can capture R-values.

```
int *a = &10; //cannot  
void foo(int &) {}  
foo(1); //cannot  
void bar(const int &) {}  
bar(1); //can
```

L-values and R-values

- Functions returning references return L-values.
- Returned R-values can be captured by const references.

```
int& getA() { return a; }  
getA() = 10;
```

```
Foo getFoo() { return Foo(); }  
const &Foo = getFoo();
```

Temporaries

- Temporary objects are R-values.
- Unintended creation of **large temporary objects** degrade performance.
- They can be ignored for primitives are small objects.

```
int bar() { return 0; }  
int foo() { return 1; }  
a = bar() + foo();
```


Thank you for helping
us help you help us all!