Summary of Memory Layout

- Most details abstracted away by IR format.
- Remember:
  - Parameters start at **fp + 4** and grow upward.
  - Locals start at **fp – 8** and grow downward.
  - Globals start at **gp + 0** and grow upward.
- You will need to write code to assign variables to these locations.

Data Representations

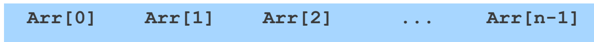
- What do different types look like in memory?
- Machine typically supports only limited types:
  - Fixed-width integers: 8-bit, 16-bit- 32-bit, signed, unsigned, etc.
  - Floating point values: 32-bit, 64-bit, 80-bit IEEE 754.
- How do we encode our object types using these types?

Encoding Primitive Types

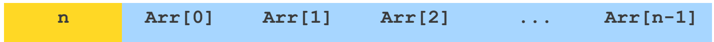
- Primitive integral types (**byte**, **char**, **short**, **int**, **long**, **unsigned**, **uint16_t**, etc.) typically map directly to the underlying machine type.
- Primitive real-valued types (**float**, **double**, **long double**) typically map directly to underlying machine type.
- Pointers typically implemented as integers holding memory addresses.
  - Size of integer depends on machine architecture; hence 32-bit compatibility mode on 64-bit machines.

Encoding Arrays

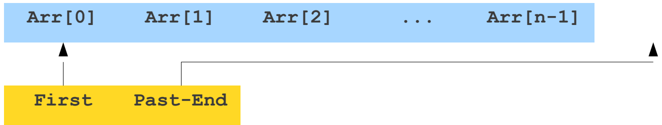
- C-style arrays: Elements laid out consecutively in memory.



- Java-style arrays: Elements laid out consecutively in memory with size information prepended.



- D-style arrays: Elements laid out consecutively in memory; array variables store pointers to first and past-the-end elements.



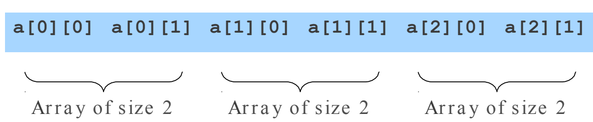
- (Which of these works well for Decaf?)

Encoding Multidimensional Arrays

- Often represented as an array of arrays.
- Shape depends on the array type used.
- C-style arrays:

  `int a[3][2];`

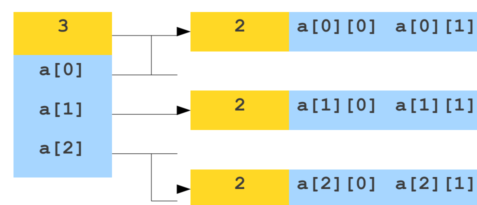
How do you know where to look for an element in an array like this?



Encoding Multidimensional Arrays

- Often represented as an array of arrays.
- Shape depends on the array type used.
- Java-style arrays:

  `int[][] a = new int [3][2];`



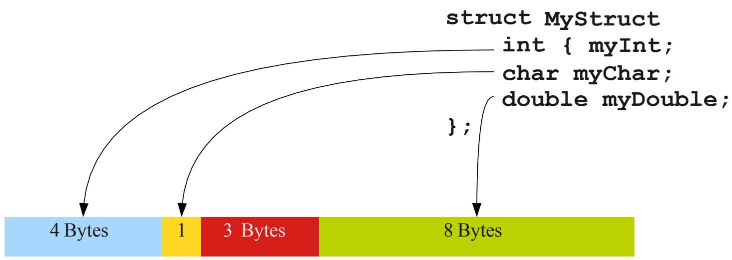
Implementing Objects

Objects are Hard

- It is difficult to build an *expressive* and *efficient* object-oriented language.
- Certain concepts are difficult to implement efficiently:
  - Dynamic dispatch (virtual functions)
  - Interfaces
  - Multiple Inheritance
  - Dynamic type checking (i.e. **`instanceof`**)
- Interfaces are so tricky to get right we won't ask you to implement them in PP4.

Encoding C-Style **structs**

- A **struct** is a type containing a collection of named values.
- Most common approach: lay each field out in the order it's declared.



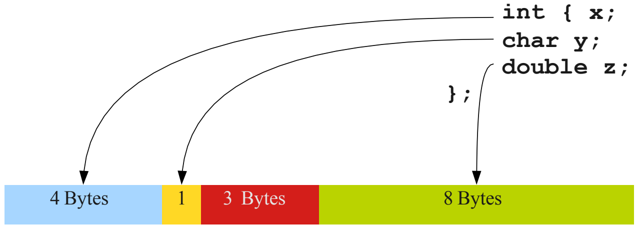
Accessing Fields

- Once an object is laid out in memory, it's just a series of bytes.
- How do we know where to look to find a particular field?

- Idea: Keep an internal table inside the compiler containing the offsets of each field.
- To look up a field, start at the base address of the object and advance forward by the appropriate offset.

Field Lookup

```
struct MyStruct
    int { x;
    char y;
    double z;
};
```




```
MyStruct* ms = new MyStruct;
ms->x = 137;  store 137  0 bytes after ms
ms->y = 'A';  store 'A'  4 bytes after ms
ms->z = 2.71  store 2.71 8 bytes after ms
```

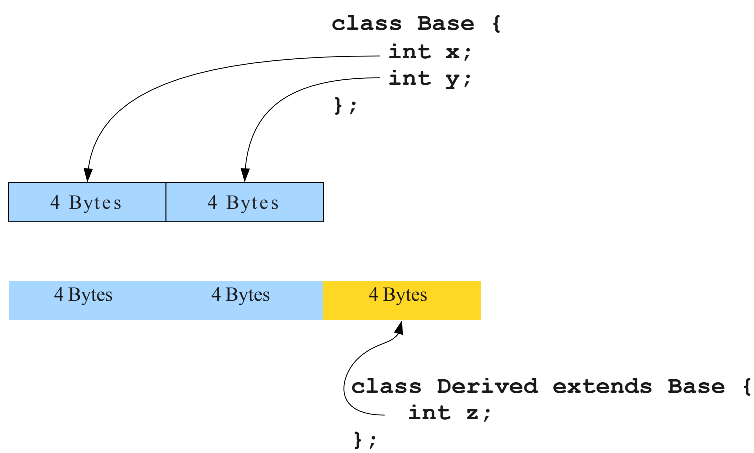
OOP without Methods

- Consider the following Decaf code:

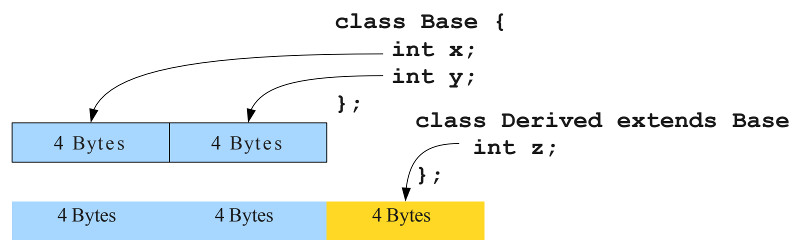
```
class Base {
    int x;
    int y;
}
class Derived extends Base {
    int z;
}
```

- What will **Derived** look like in memory?

Memory Layouts with Inheritance



Field Lookup With Inheritance




```
Base ms = new Base;
ms.x = 137;      store 137 0 bytes after ms
ms.y = 42;       store 42  4 bytes after ms

Base ms = new Derived;
ms.x = 137;      store 137 0 bytes after ms
ms.y = 42;       store 42  4 bytes after ms
```

Single Inheritance in Decaf

- The memory layout for a class D that extends B is given by the memory layout for B followed by the memory layout for the members of D.
  - Actually a bit more complex; we'll see why later.
- Rationale: A pointer of type B pointing at a D object still sees the B object at the beginning.
- Operations done on a D object through the B reference guaranteed to be safe; no need to check what B points at dynamically.

What About Member Functions?

- Member functions are mostly like regular functions, but with two complications:
  - How do we know what receiver object to use?
  - How do we know which function to call at runtime (dynamic dispatch)?

this is Tricky

- Inside a member function, the name **this** refers to the current receiver object.
- This information (pun intended) needs to be communicated into the function.
- **Idea:** Treat **this** as an implicit first parameter.
- Every n-argument member function is really an (n+1)-argument member function whose first parameter is the **this** pointer.

this is Clever

```
class MyClass {
    int x;
    void myFunction(int arg) {
        this.x = arg;
    }
}

MyClass m = new MyClass;
m.myFunction(137);
```

this is Clever

```
class MyClass {
    int x;
}
void MyClass_myFunction(MyClass this, int arg){
    this.x = arg;
}

MyClass m = new MyClass;
m.myFunction(137);
```

this is Clever

```
class MyClass {
    int x;
}
void MyClass_myFunction(MyClass this, int arg){
    this.x = arg;
}

MyClass m = new MyClass;
MyClass_myFunction(m, 137);
```

`this` Rules

- When generating code to call a member function, remember to pass some object as the **`this`** parameter representing the receiver object.
- Inside of a member function, treat **`this`** as just another parameter to the member function.
- When implicitly referring to a field of **`this`**, use this extra parameter as the object in which the field should be looked up.

Implementing Dynamic Dispatch

- **Dynamic dispatch** means calling a function at runtime based on the dynamic type of an object, rather than its static type.
- How do we set up our runtime environment so that we can efficiently support this?

An Initial Idea

- At compile-time, get a list of every defined class.
- To compile a dynamic dispatch, emit IR code for the following logic:

```
if (the object has type A)
    call A's version of the function
else if (the object has type B)
    call B's version of the function
…
else if (the object has type N)
    call N's version of the function.
```

Analyzing our Approach

- This previous idea has several serious problems.
- What are they?
- **It's slow.**
  - Number of checks is O(C), where C is the number of classes the dispatch might refer to.
  - Gets slower the more classes there are.
- **It's infeasible in most languages.**
  - What if we link across multiple source files?
  - What if we support dynamic class loading?

An Observation

- When laying out fields in an object, we gave every field an offset.
- Derived classes have the base class fields in the same order at the beginning.

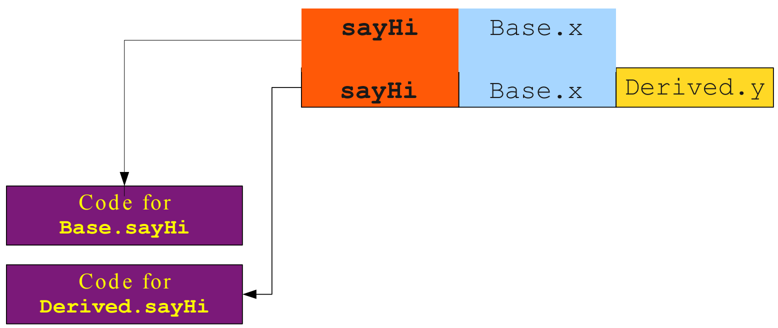
Layout of **Base**	Base.x	Base.y	
Layout of **Derived**	Base.x	Base.y	Derived.z

- Can we do something similar with functions?

Virtual Function Tables

```
class Base {
    int x;
    void sayHi() {
        Print("Base");
    }
}
```

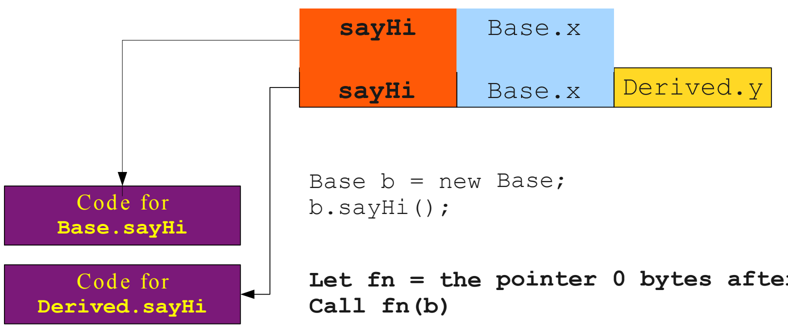
```
class Derived extends Base {
    int y;
    void sayHi() {
        Print("Derived");
    }
}
```



Virtual Function Tables

```
class Base {
    int x;
    void sayHi() {
        Print("Base");
    }
}
```

```
class Derived extends Base {
   int y;
   void sayHi() {
       Print("Derived");
   }
}
```



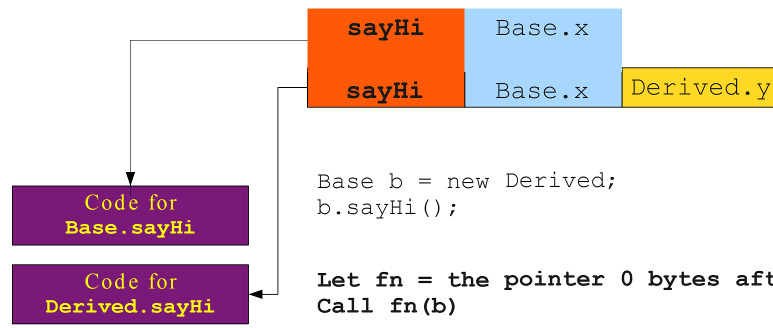

```
Base b = new Base;
b.sayHi();
```

Let fn = the pointer 0 bytes after b
Call fn(b)

Virtual Function Tables

```
class Base {
    int x;
    void sayHi() {
        Print("Base");
    }
}
```

```
class Derived extends Base {
   int y;
   void sayHi() {
       Print("Derived");
   }
}
```




```
Base b = new Derived;
b.sayHi();
```

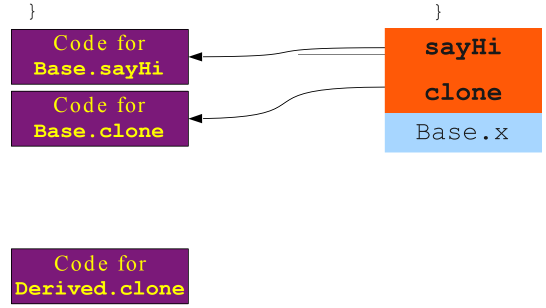
Let fn = the pointer 0 bytes after b
Call fn(b)

More Virtual Function Tables

```
class Base {                    class Derived extends Base
    int x;                        { int y;
    void sayHi() {
        Print("Hi Mom!");
    }
    Base clone() {                Derived clone() {
        return new Base;              return new Derived;
    }                             }
}                               }
```

More Virtual Function Tables

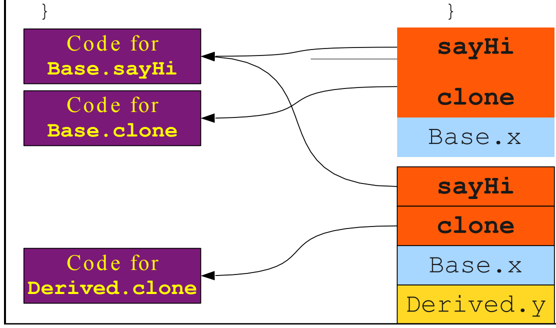
```
class Base {                    class Derived extends Base
    int x;                        { int y;
    void sayHi() {
        Print("Hi Mom!");
    }
    Base clone() {                Derived clone() {
        return new Base;              return new Derived;
    }                             }
}                               }
```



More Virtual Function Tables

```
class Base {
    int x;
    void sayHi() {
        Print("Hi Mom!");
    }
    Base clone() {
        return new Base;
    }
}
```

```
class Derived extends Base
   { int y;


   Derived clone() {
       return new Derived;
   }
}
```



Virtual Function Tables

- A **virtual function table** (or **vtable**) is an array of pointers to the member function implementations for a particular class.
- To invoke a member function:
  - Determine (statically) its index in the vtable.
  - Follow the pointer at that index in the object's vtable to the code for the function.
  - Invoke that function.

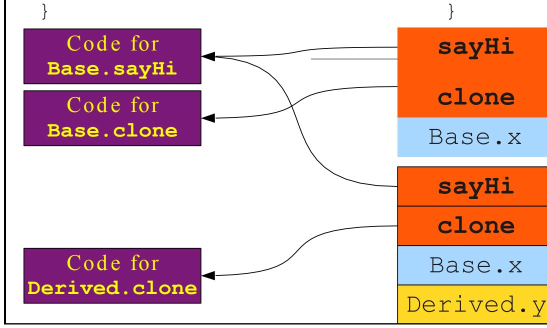
Analyzing our Approach

- Advantages:
  - Time to determine function to call is O(1).
  - (and a good O(1) too!)
- What are the disadvantages?
- **Object sizes are larger.**
  - Each object needs to have space for O(*M*) pointers.
- **Object creation is slower.**
  - Each new object needs to have O(*M*) pointers set, where *M* is the number of member functions.

A Common Optimization

```
class Base {
    int x;
    void sayHi() {
        Print("Base");
    }
    Base clone() {
        return new Base;
    }
}
```

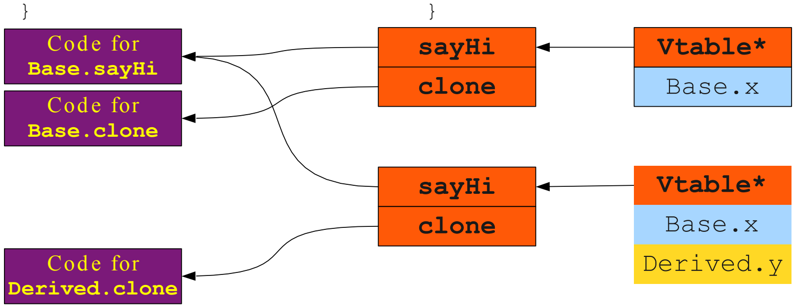
```
class Derived extends Base
   { int y;


   Derived clone() {
       return new Derived;
   }
}
```



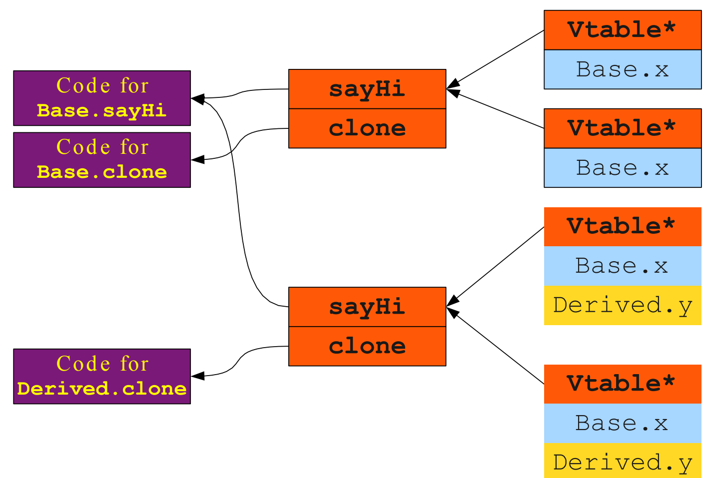
A Common Optimization

```
class Base {
    int x;
    void sayHi() {
        Print("Base");
    }
    Base clone() {
        return new Base;
    }
}
```

```
class Derived extends Base
    { int y;

    Derived clone() {
        return new Derived;
    }
}
```



Objects in Memory



Dynamic Dispatch in O(1)

- Create a single instance of the vtable for each class.
- Have each object store a pointer to the vtable.
- Can follow the pointer to the table in O(1).
- Can index into the table in O(1).
- Can set the vtable pointer of a new object in O(1).
- Increases the size of each object by O(1).
- **This is the solution used in most C++ and Java implementations.**

Vtable Requirements

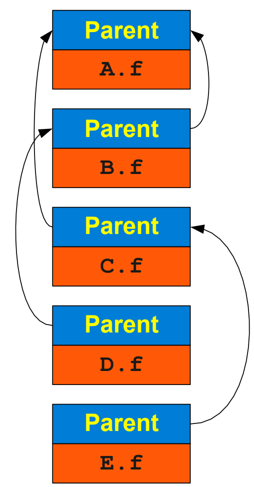
- We've made implicit assumptions about our language that allow vtables to work correctly.
- What are they?
- **Method calls known statically.**
  - We can determine at compile-time which methods are intended at each call (even if we're not sure which method is ultimately invoked).
- **Single inheritance.**
  - Don't need to worry about building a single vtable for multiple different classes.

Dynamic Type Checks

- Many languages require some sort of dynamic type checking.
  - Java's **instanceof**, C++'s **dynamic_cast**, any dynamically-typed language.
- May want to determine whether the dynamic type is *convertible* to some other type, not whether the type is *equal*.
- How can we implement this?

A Pretty Good Approach

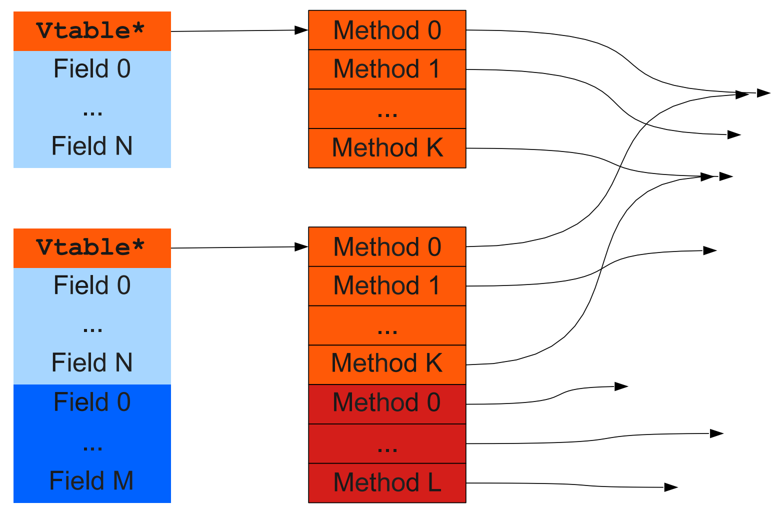
```
class A {
   void f() {}
}        extends A {

class B
   void f() {}
}
class C extends A {

   void f() {}
}
class D extends B {
   void f() {}
}

class E extends C {
   void f() {}
}
```



Simple Dynamic Type Checking

- Have each object's vtable store a pointer to its base class.
- To check if an object is convertible to type *S* at runtime, follow the pointers embedded in the object's vtable upward until we find *S* or reach a type with no parent.
- Runtime is O(d), where d is the depth of the class in the hierarchy.

A Reminder: Object Layout



TAC for Objects, Part I

```
class A {
    void fn(int x) {
        int y;
        y = x;
    }
}

int main()
    {  A a;
    a.fn(137);
}
```

```
_A.fn:
    BeginFunc 4;
    y = x;
    EndFunc;

main:
    BeginFunc 8;
    _t0 = 137;
    PushParam _t0;
    PushParam a;
    LCall _A.fn;
    PopParams 8;
    EndFunc;
```

TAC for Objects, Part II

```
class A {
    int y;
    int z;
    void fn(int x) {
        y = x;
        x = z;
    }
}

int main()
    {  A a;
    a.fn(137);
}
```

```
_A.fn:
    BeginFunc 4;
    *(this + 4) = x;
    x = *(this + 8);
    EndFunc;

main:
    BeginFunc 8;
    _t0 = 137;
    PushParam _t0;
    PushParam a;
    LCall _A.fn;
    PopParams 8;
    EndFunc;
```

Memory Access in TAC

- Extend our simple assignments with memory accesses:
  - $\texttt{var}_1$ = $\texttt{*var}_2$
  - $\texttt{var}_1$ = $\texttt{*(var}_2$ + constant)
  - $\texttt{*var}_1$ = $\texttt{var}_2$
  - $\texttt{*(var}_1$ + constant) =
- $\texttt{var}_2$

You will need to translate field accesses into relative memory accesses.

TAC for Objects, Part III

```
class Base
  void { hi()
    Print("Base");
  }
}

class Derived extends Base{
  void hi() {
    Print("Derived");
  }
}

int main() {
    Base b;
    b = new Derived;
    b.hi();
}
```

```
_Base.hi:
    BeginFunc 4;
    _t0 = "Base";
    PushParam _t0;
    LCall _PrintString;
    PopParams 4;
    EndFunc;
Vtable Base = _Base.hi,
    ;

_Derived.hi:
    BeginFunc 4;
    _t0 = "Derived";
    PushParam _t0;
    LCall _PrintString;
    PopParams 4;
    EndFunc;
Vtable Derived = _Derived.hi,
    ;
```

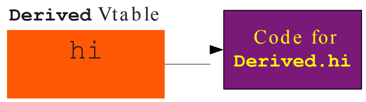
TAC for Objects, Part III

```
class Base
  void { hi()
    Print("Base");
  }
}

class Derived extends Base{
  void hi() {
    Print("Derived");
  }
}

int main() {
    Base b;
    b = new Derived;
    b.hi();
}
```

```
main:
    BeginFunc 20;
    _t0 = 4;
    PushParam _t0;
    b = LCall _Alloc;
    PopParams 4;
    _t1 = Derived;
    *b = _t1;
    _t2 = *b;
    _t3 = *_t2;
    PushParam b;
    ACall _t3;
    PopParams 4;
    EndFunc;
```

What's going on here?

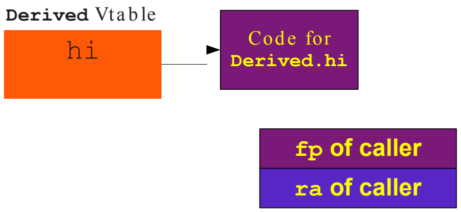
Dissecting TAC




```
int main() {
    Base b;
    b = new Derived;
    b.hi();
}
```

```
main:
    BeginFunc 20;
    _t0 = 4;
    PushParam _t0;
    b = LCall _Alloc;
    PopParams 4;
    _t1 = Derived;
    *b = _t1;
    _t2 = *b;
    _t3 = *_t2;
    PushParam b;
    ACall _t3;
    PopParams 4;
    EndFunc;
```

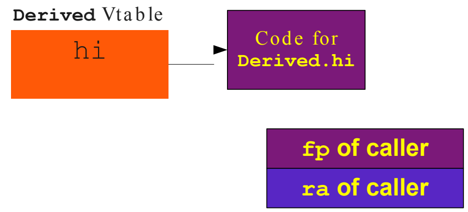
Dissecting TAC




```
int main() {
    Base b;
    b = new Derived;
    b.hi();
}
```

```
main:
    BeginFunc 20;
    _t0 = 4;
    PushParam _t0;
    b = LCall _Alloc;
    PopParams 4;
    _t1 = Derived;
    *b = _t1;
    _t2 = *b;
    _t3 = *_t2;
    PushParam b;
    ACall _t3;
    PopParams 4;
    EndFunc;
```

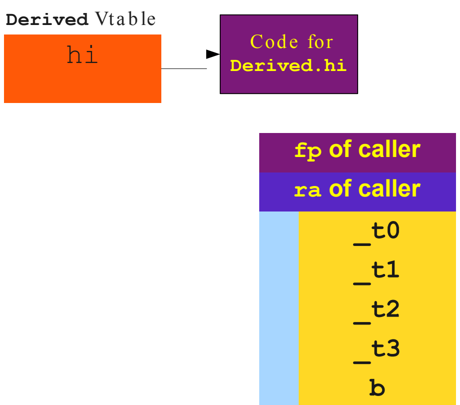
Dissecting TAC




```
int main() {
    Base b;
    b = new Derived;
    b.hi();
}
```

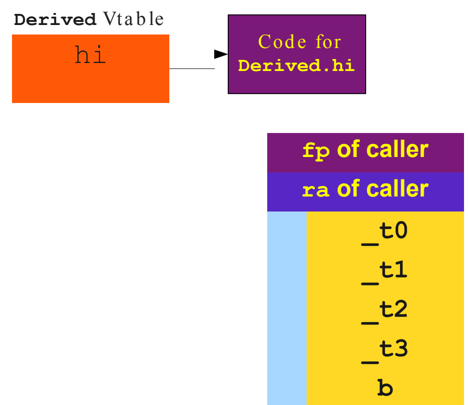
```
main:
    BeginFunc 20;
    _t0 = 4;
    PushParam _t0;
    b = LCall _Alloc;
    PopParams 4;
    _t1 = Derived;
    *b = _t1;
    _t2 = *b;
    _t3 = *_t2;
    PushParam b;
    ACall _t3;
    PopParams 4;
    EndFunc;
```

Dissecting TAC



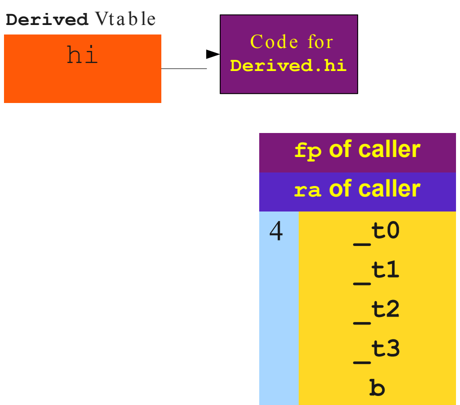

```
int main() {
     Base b;
     b = new Derived;
     b.hi();
}
main:
     BeginFunc 20;
     _t0 = 4;
     PushParam _t0;
     b = LCall _Alloc;
     PopParams 4;
     _t1 = Derived;
     *b = _t1;
     _t2 = *b;
     _t3 = *_t2;
     PushParam b;
     ACall _t3;
     PopParams 4;
     EndFunc;
```

Dissecting TAC




```
int main() {
     Base b;
     b = new Derived;
     b.hi();
}
main:
     BeginFunc 20;
     _t0 = 4;
     PushParam _t0;
     b = LCall _Alloc;
     PopParams 4;
     _t1 = Derived;
     *b = _t1;
     _t2 = *b;
     _t3 = *_t2;
     PushParam b;
     ACall _t3;
     PopParams 4;
     EndFunc;
```

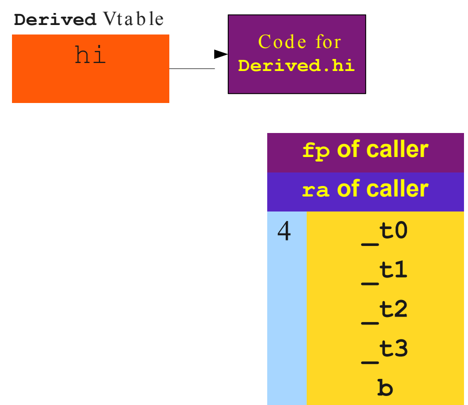
Dissecting TAC




```
int main() {
    Base b;
    b = new Derived;
    b.hi();
}
```

```
main:
    BeginFunc 20;
    _t0 = 4;
    PushParam _t0;
    b = LCall _Alloc;
    PopParams 4;
    _t1 = Derived;
    *b = _t1;
    _t2 = *b;
    _t3 = *_t2;
    PushParam b;
    ACall _t3;
    PopParams 4;
    EndFunc;
```

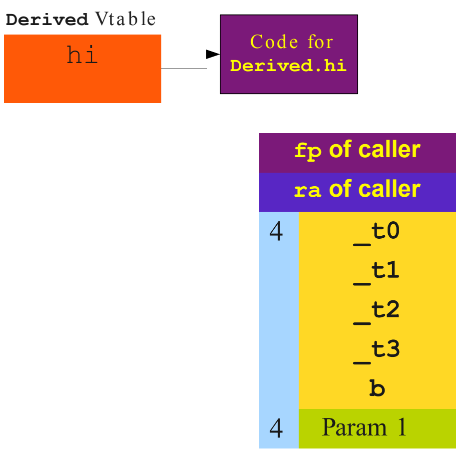
Dissecting TAC




```
int main() {
    Base b;
    b = new Derived;
    b.hi();
}
```

```
main:
    BeginFunc 20;
    _t0 = 4;
    PushParam _t0;
    b = LCall _Alloc;
    PopParams 4;
    _t1 = Derived;
    *b = _t1;
    _t2 = *b;
    _t3 = *_t2;
    PushParam b;
    ACall _t3;
    PopParams 4;
    EndFunc;
```

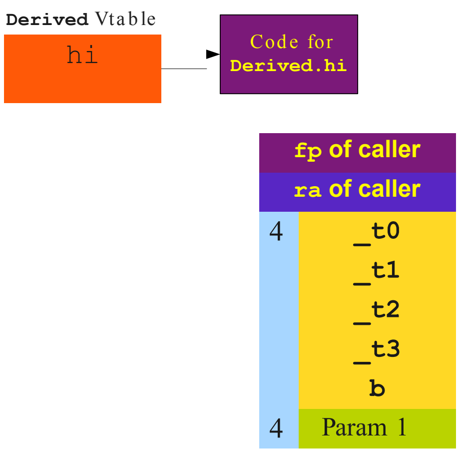
Dissecting TAC




```
int main() {
     Base b;
     b = new Derived;
     b.hi();
}
```

```
main:
     BeginFunc 20;
     _t0 = 4;
     PushParam _t0;
     b = LCall _Alloc;
     PopParams 4;
     _t1 = Derived;
     *b = _t1;
     _t2 = *b;
     _t3 = *_t2;
     PushParam b;
     ACall _t3;
     PopParams 4;
     EndFunc;
```

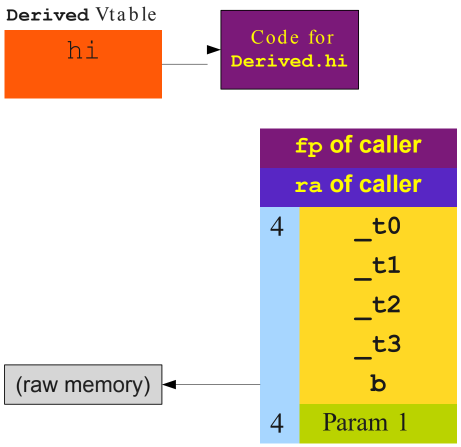
Dissecting TAC




```
int main() {
     Base b;
     b = new Derived;
     b.hi();
}
```

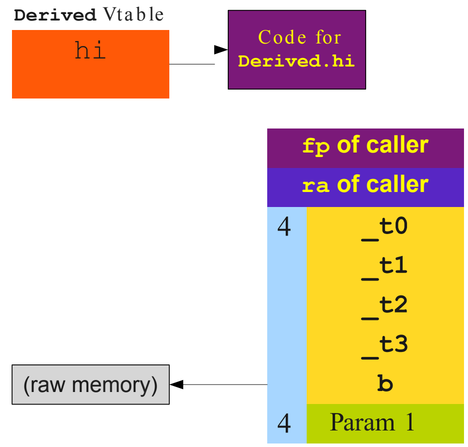
```
main:
     BeginFunc 20;
     _t0 = 4;
     PushParam _t0;
     b = LCall _Alloc;
     PopParams 4;
     _t1 = Derived;
     *b = _t1;
     _t2 = *b;
     _t3 = *_t2;
     PushParam b;
     ACall _t3;
     PopParams 4;
     EndFunc;
```

Dissecting TAC



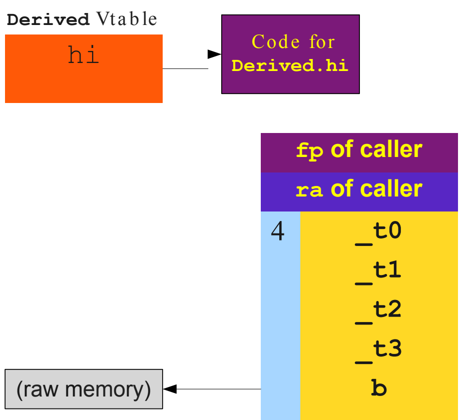

```
int main() {
    Base b;
    b = new Derived;
    b.hi();
}

main:
    BeginFunc 20;
    _t0 = 4;
    PushParam _t0;
    b = LCall _Alloc;
    PopParams 4;
    _t1 = Derived;
    *b = _t1;
    _t2 = *b;
    _t3 = *_t2;
    PushParam b;
    ACall _t3;
    PopParams 4;
    EndFunc;
```

Dissecting TAC



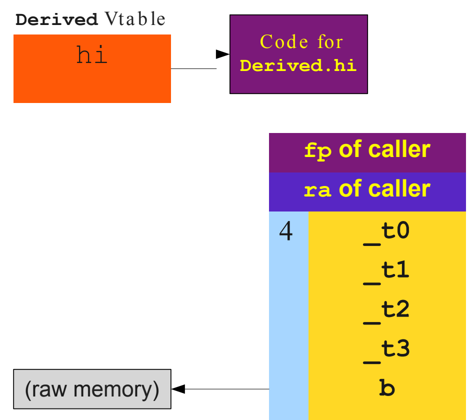

```
int main() {
    Base b;
    b = new Derived;
    b.hi();
}

main:
    BeginFunc 20;
    _t0 = 4;
    PushParam _t0;
    b = LCall _Alloc;
    PopParams 4;
    _t1 = Derived;
    *b = _t1;
    _t2 = *b;
    _t3 = *_t2;
    PushParam b;
    ACall _t3;
    PopParams 4;
    EndFunc;
```

Dissecting TAC



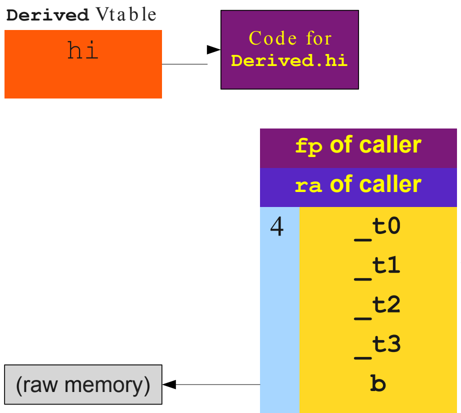

```
int main() {
    Base b;
    b = new Derived;
    b.hi();
}

main:
    BeginFunc 20;
    _t0 = 4;
    PushParam _t0;
    b = LCall _Alloc;
    PopParams 4;
    _t1 = Derived;
    *b = _t1;
    _t2 = *b;
    _t3 = *_t2;
    PushParam b;
    ACall _t3;
    PopParams 4;
    EndFunc;
```

Dissecting TAC




```
int main() {
    Base b;
    b = new Derived;
    b.hi();
}

main:
    BeginFunc 20;
    _t0 = 4;
    PushParam _t0;
    b = LCall _Alloc;
    PopParams 4;
    _t1 = Derived;
    *b = _t1;
    _t2 = *b;
    _t3 = *_t2;
    PushParam b;
    ACall _t3;
    PopParams 4;
    EndFunc;
```

Dissecting TAC

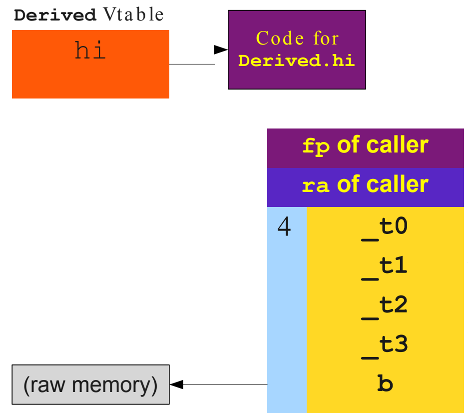



```
int main() {
     Base b;
     b = new Derived;
     b.hi();
}
```

```
main:
     BeginFunc 20;
     _t0 = 4;
     PushParam _t0;
     b = LCall _Alloc;
     PopParams 4;
     _t1 = Derived;
     *b = _t1;
     _t2 = *b;
     _t3 = *_t2;
     PushParam b;
     ACall _t3;
     PopParams 4;
     EndFunc;
```

Allocate Object

Dissecting TAC

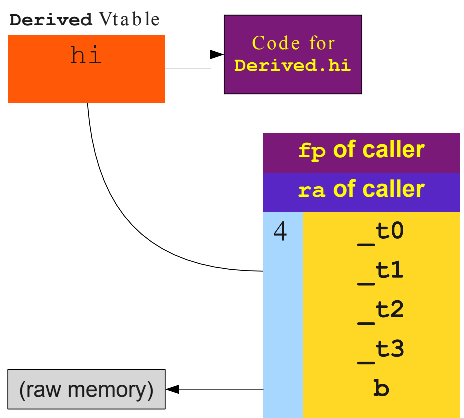



```
int main() {
     Base b;
     b = new Derived;
     b.hi();
}
```

```
main:
     BeginFunc 20;
     _t0 = 4;
     PushParam _t0;
     b = LCall _Alloc;
     PopParams 4;
     _t1 = Derived;
     *b = _t1;
     _t2 = *b;
     _t3 = *_t2;
     PushParam b;
     ACall _t3;
     PopParams 4;
     EndFunc;
```

Allocate Object

Dissecting TAC

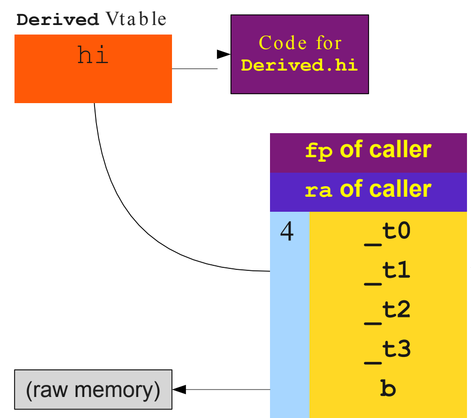



```
int main() {
     Base b;
     b = new Derived;
     b.hi();
}
```

Allocate Object

```
main:
     BeginFunc 20;
     _t0 = 4;
     PushParam _t0;
     b = LCall _Alloc;
     PopParams 4;
     _t1 = Derived;
     *b = _t1;
     _t2 = *b;
     _t3 = *_t2;
     PushParam b;
     ACall _t3;
     PopParams 4;
     EndFunc;
```

Dissecting TAC

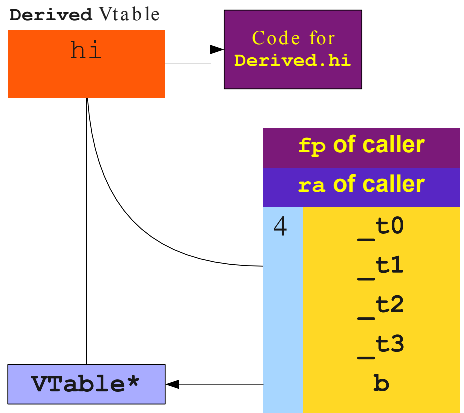



```
int main() {
     Base b;
     b = new Derived;
     b.hi();
}
```

Allocate Object

```
main:
     BeginFunc 20;
     _t0 = 4;
     PushParam _t0;
     b = LCall _Alloc;
     PopParams 4;
     _t1 = Derived;
     *b = _t1;
     _t2 = *b;
     _t3 = *_t2;
     PushParam b;
     ACall _t3;
     PopParams 4;
     EndFunc;
```

Dissecting TAC

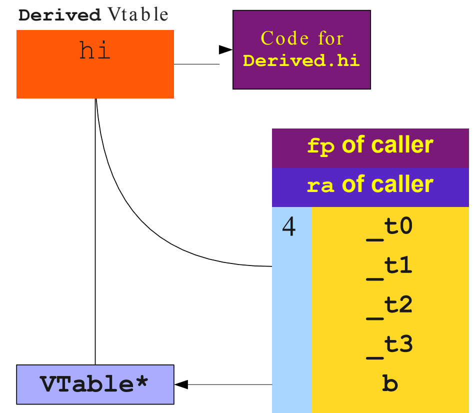



```
int main() {
     Base b;
     b = new Derived;
     b.hi();
}
```

```
main:
     BeginFunc 20;
     _t0 = 4;
     PushParam _t0;
     b = LCall _Alloc;
     PopParams 4;
     _t1 = Derived;
     *b = _t1;
     _t2 = *b;
     _t3 = *_t2;
     PushParam b;
     ACall _t3;
     PopParams 4;
     EndFunc;
```

Allocate Object

Dissecting TAC



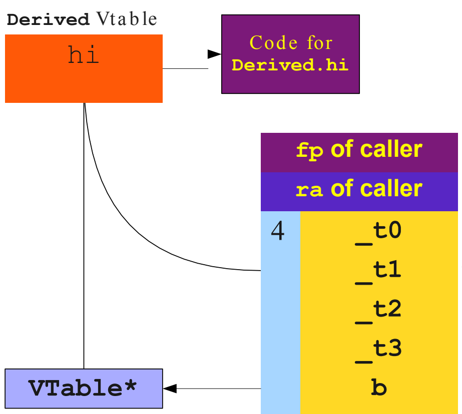

```
int main() {
     Base b;
     b = new Derived;
     b.hi();
}
```

```
main:
     BeginFunc 20;
     _t0 = 4;
     PushParam _t0;
     b = LCall _Alloc;
     PopParams 4;
     _t1 = Derived;
     *b = _t1;
     _t2 = *b;
     _t3 = *_t2;
     PushParam b;
     ACall _t3;
     PopParams 4;
     EndFunc;
```

Allocate Object

Set Vtable

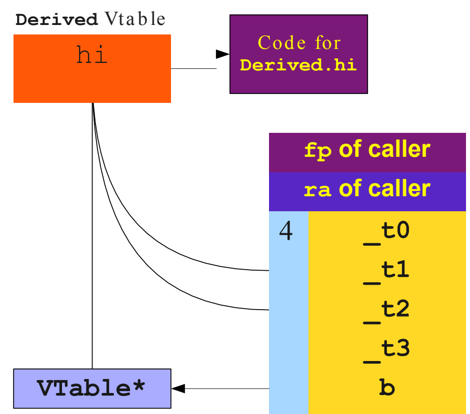
Dissecting TAC




```
int main() {
     Base b;
     b = new Derived;
     b.hi();
}
main:
     BeginFunc 20;
     _t0 = 4;
     PushParam _t0;
     b = LCall _Alloc;
     PopParams 4;
     _t1 = Derived;
     *b = _t1;
     _t2 = *b;
     _t3 = *_t2;
     PushParam b;
     ACall _t3;
     PopParams 4;
     EndFunc;
```

Allocate Object
Set Vtable

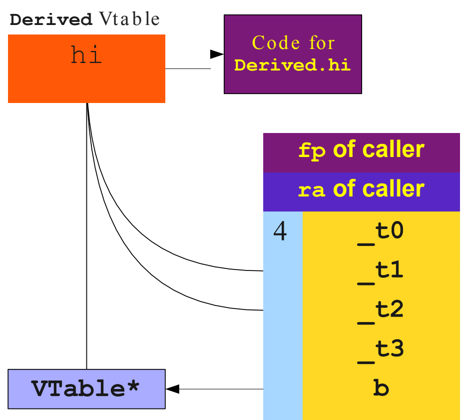
Dissecting TAC




```
int main() {
     Base b;
     b = new Derived;
     b.hi();
}
main:
     BeginFunc 20;
     _t0 = 4;
     PushParam _t0;
     b = LCall _Alloc;
     PopParams 4;
     _t1 = Derived;
     *b = _t1;
     _t2 = *b;
     _t3 = *_t2;
     PushParam b;
     ACall _t3;
     PopParams 4;
     EndFunc;
```

Allocate Object
Set Vtable

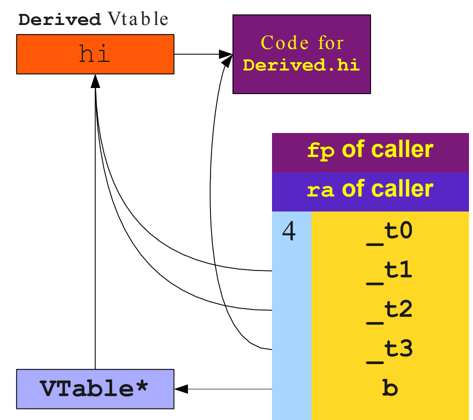
Dissecting TAC




```
int main() {
    Base b;
    b = new Derived;
    b.hi();
}
```

```
main:
                BeginFunc 20;
Allocate   {    _t0 = 4;
Object     {    PushParam _t0;
           {    b = LCall _Alloc;
           {    PopParams 4;
Set        {    _t1 = Derived;
Vtable     {    *b = _t1;
                _t2 = *b;
                _t3 = *_t2;
                PushParam b;
                ACall _t3;
                PopParams 4;
                EndFunc;
```

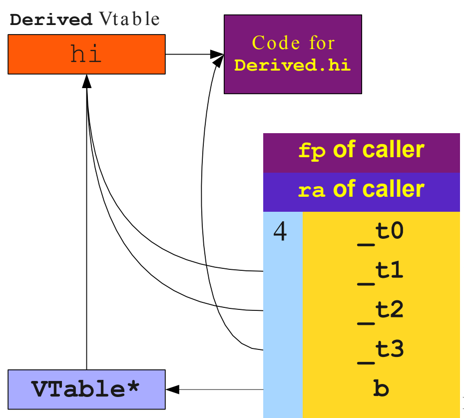
Dissecting TAC




```
int main() {
    Base b;
    b = new Derived;
    b.hi();
}
```

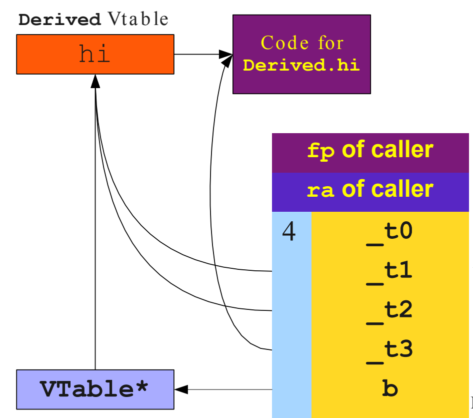
```
main:
                BeginFunc 20;
Allocate   {    _t0 = 4;
Object     {    PushParam _t0;
           {    b = LCall _Alloc;
           {    PopParams 4;
Set        {    _t1 = Derived;
Vtable     {    *b = _t1;
                _t2 = *b;
                _t3 = *_t2;
                PushParam b;
                ACall _t3;
                PopParams 4;
                EndFunc;
```

Dissecting TAC



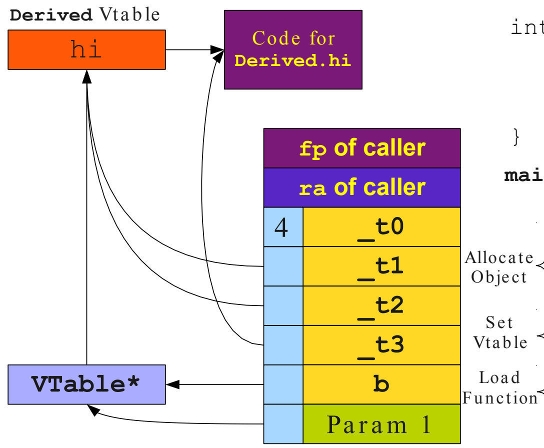

```
int main() {
     Base b;
     b = new Derived;
     b.hi();
}
main:
     BeginFunc 20;
Allocate Object
     _t0 = 4;
     PushParam _t0;
     b = LCall _Alloc;
     PopParams 4;
Set Vtable
     _t1 = Derived;
     *b = _t1;
Load Function
     _t2 = *b;
     _t3 = *_t2;
     PushParam b;
     ACall _t3;
     PopParams 4;
     EndFunc;
```

Dissecting TAC



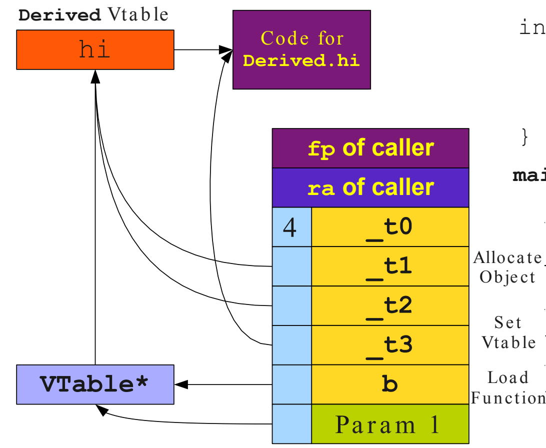

```
int main() {
     Base b;
     b = new Derived;
     b.hi();
}
main:
     BeginFunc 20;
Allocate Object
     _t0 = 4;
     PushParam _t0;
     b = LCall _Alloc;
     PopParams 4;
Set Vtable
     _t1 = Derived;
     *b = _t1;
Load Function
     _t2 = *b;
     _t3 = *_t2;
     PushParam b;
     ACall _t3;
     PopParams 4;
     EndFunc;
```

Dissecting TAC



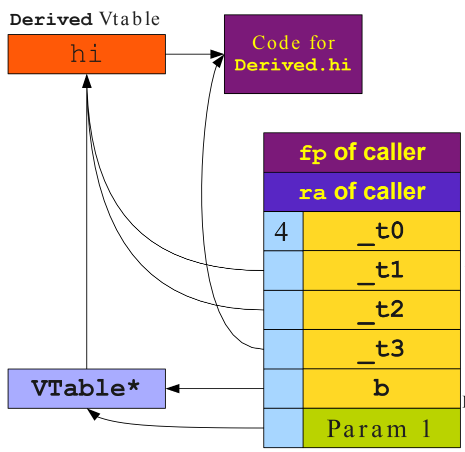

```
int main() {
     Base b;
     b = new Derived;
     b.hi();
}
main:
     BeginFunc 20;
     _t0 = 4;
     PushParam _t0;
     b = LCall _Alloc;
     PopParams 4;
     _t1 = Derived;
     *b = _t1;
     _t2 = *b;
     _t3 = *_t2;
     PushParam b;
     ACall _t3;
     PopParams 4;
     EndFunc;
```

Dissecting TAC




```
int main() {
     Base b;
     b = new Derived;
     b.hi();
}
main:
     BeginFunc 20;
     _t0 = 4;
     PushParam _t0;
     b = LCall _Alloc;
     PopParams 4;
     _t1 = Derived;
     *b = _t1;
     _t2 = *b;
     _t3 = *_t2;
     PushParam b;
     ACall _t3;
     PopParams 4;
     EndFunc;
```

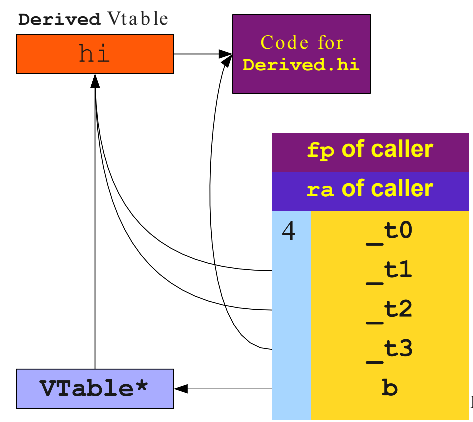
Dissecting TAC




```
int main() {
     Base b;
     b = new Derived;
     b.hi();
}
```

```
main:
              BeginFunc 20;
Allocate      _t0 = 4;
Object        PushParam _t0;
              b = LCall _Alloc;
              PopParams 4;
Set           _t1 = Derived;
Vtable        *b = _t1;
Load          _t2 = *b;
Function      _t3 = *_t2;
              PushParam b;
              ACall _t3;
              PopParams 4;
              EndFunc;
```

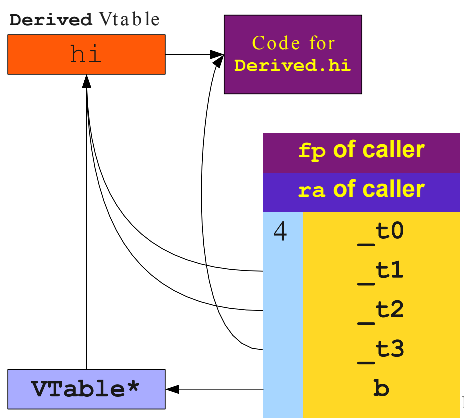
Dissecting TAC




```
int main() {
     Base b;
     b = new Derived;
     b.hi();
}
```

```
main:
              BeginFunc 20;
Allocate      _t0 = 4;
Object        PushParam _t0;
              b = LCall _Alloc;
              PopParams 4;
Set           _t1 = Derived;
Vtable        *b = _t1;
Load          _t2 = *b;
Function      _t3 = *_t2;
              PushParam b;
              ACall _t3;
              PopParams 4;
              EndFunc;
```

Dissecting TAC




```
int main() {
     Base b;
     b = new Derived;
     b.hi();
}
```

```
main:
          BeginFunc 20;
          _t0 = 4;
Allocate  PushParam _t0;
Object    b = LCall _Alloc;
          PopParams 4;
Set       _t1 = Derived;
Vtable    *b = _t1;
Load      _t2 = *b;
Function  _t3 = *_t2;
          PushParam b;
          ACall _t3;
          PopParams 4;
          EndFunc;
```

OOP in TAC

- The address of an object's vtable can be referenced via the name assigned to the vtable (usually the object name).
  - e.g. **`_t0 = Base;`**
- When creating objects, you must remember to set the object's vtable pointer or any method call will cause a crash at runtime.
- The **`ACall`** instruction can be used to call a method given a pointer to the first instruction.