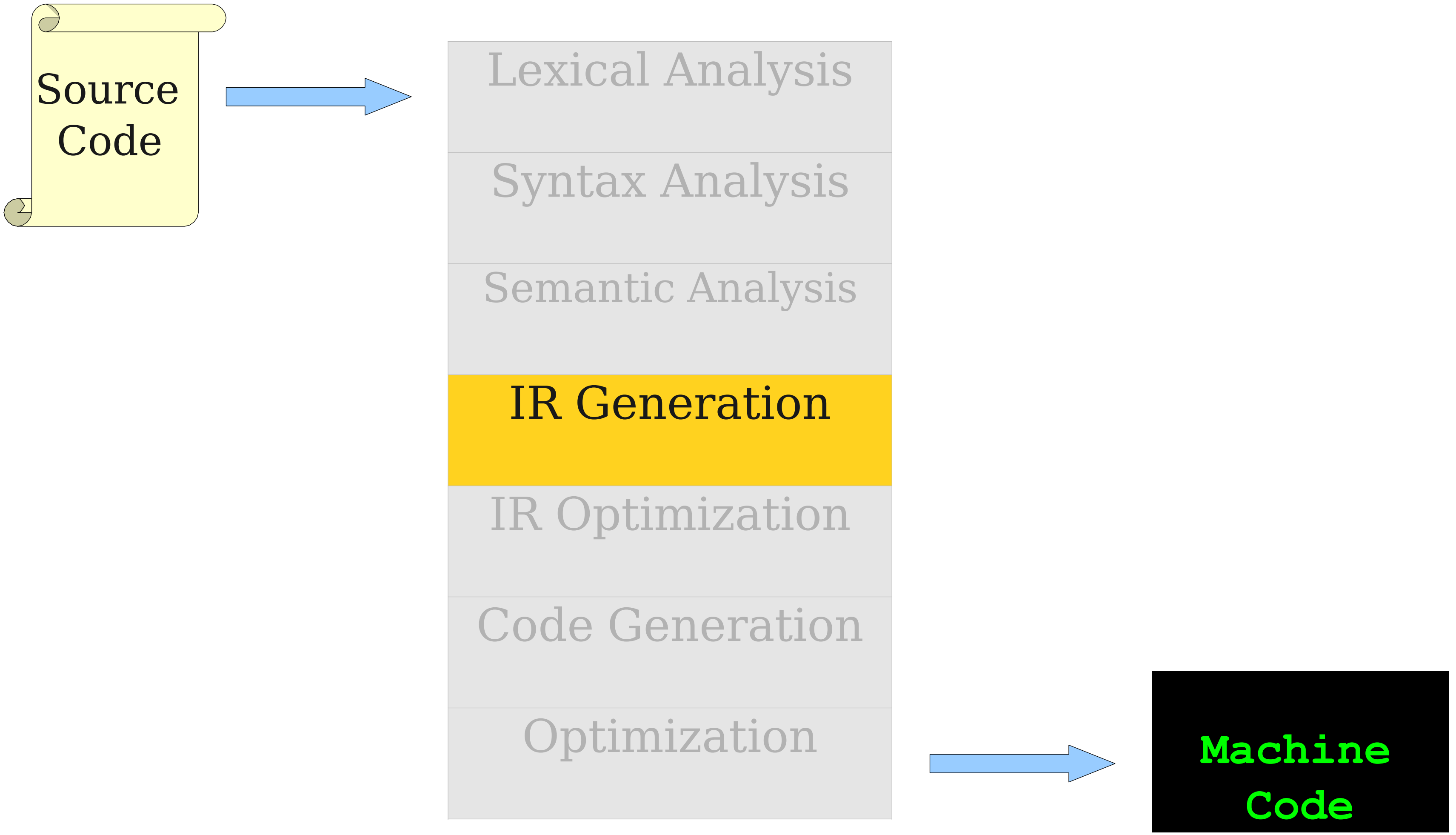
بسم الله الرحمن الرحيم

Semantic Analysis, Runtime environment (2)

Runtime Environments

Part II

Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Implementing Objects

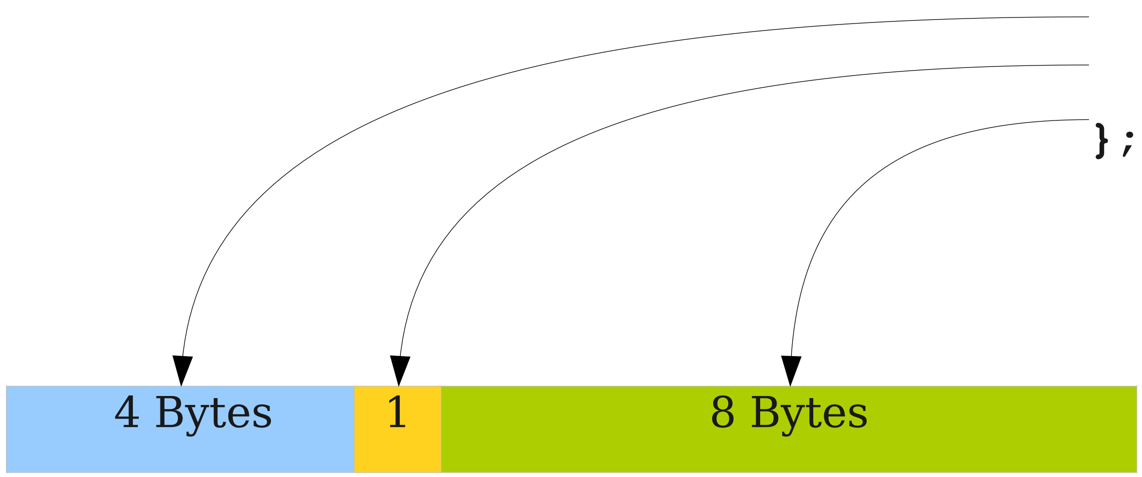
Implementing Object-oriented Features

- It is hard
- Dynamic dispatch (virtual functions)
- Interfaces
- Multiple Inheritance
- Dynamic type checking (i.e. instanceof)

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.

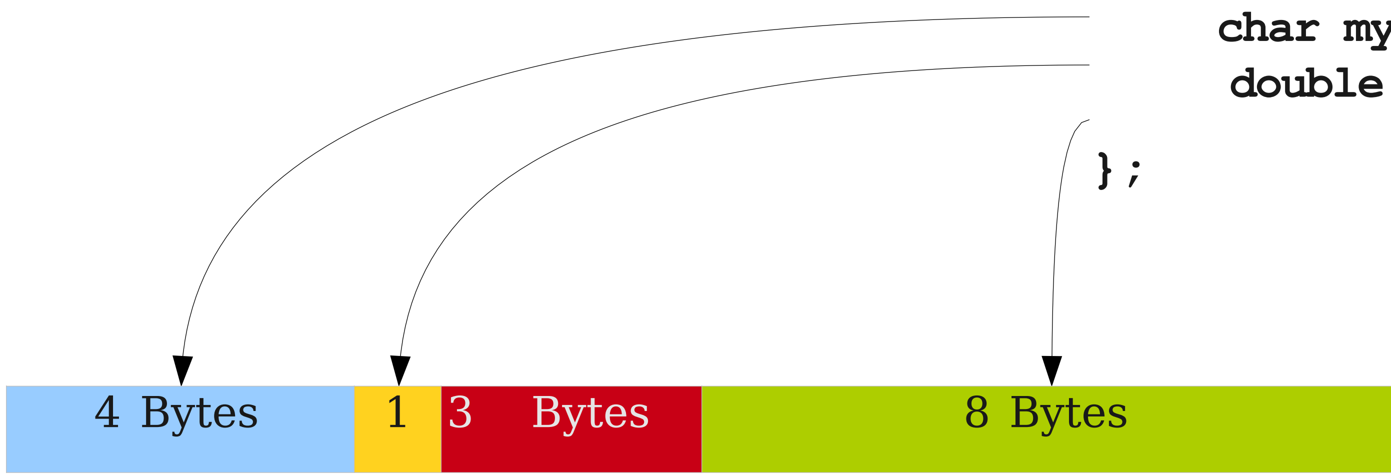
```
struct MyStruct {
      int myInt;
      char myChar;
      double myDouble;
};
```



Encoding C-Style **`structs`**

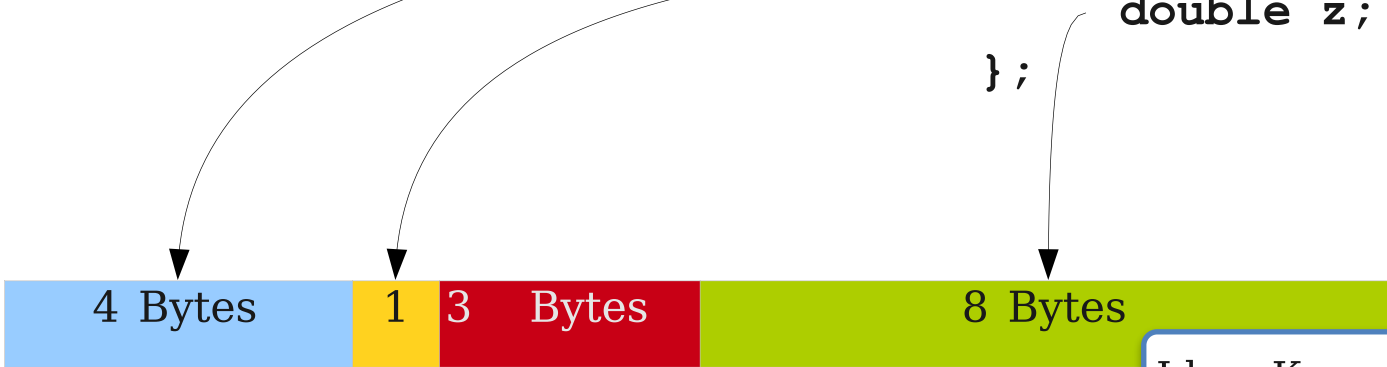
- A **`struct`** is a type containing a collection of named values.
- Most common approach: lay each field out in the order it's declared.

```
struct MyStruct {
     int myInt;
     char myChar;
     double myDouble;
};
```



Field Lookup

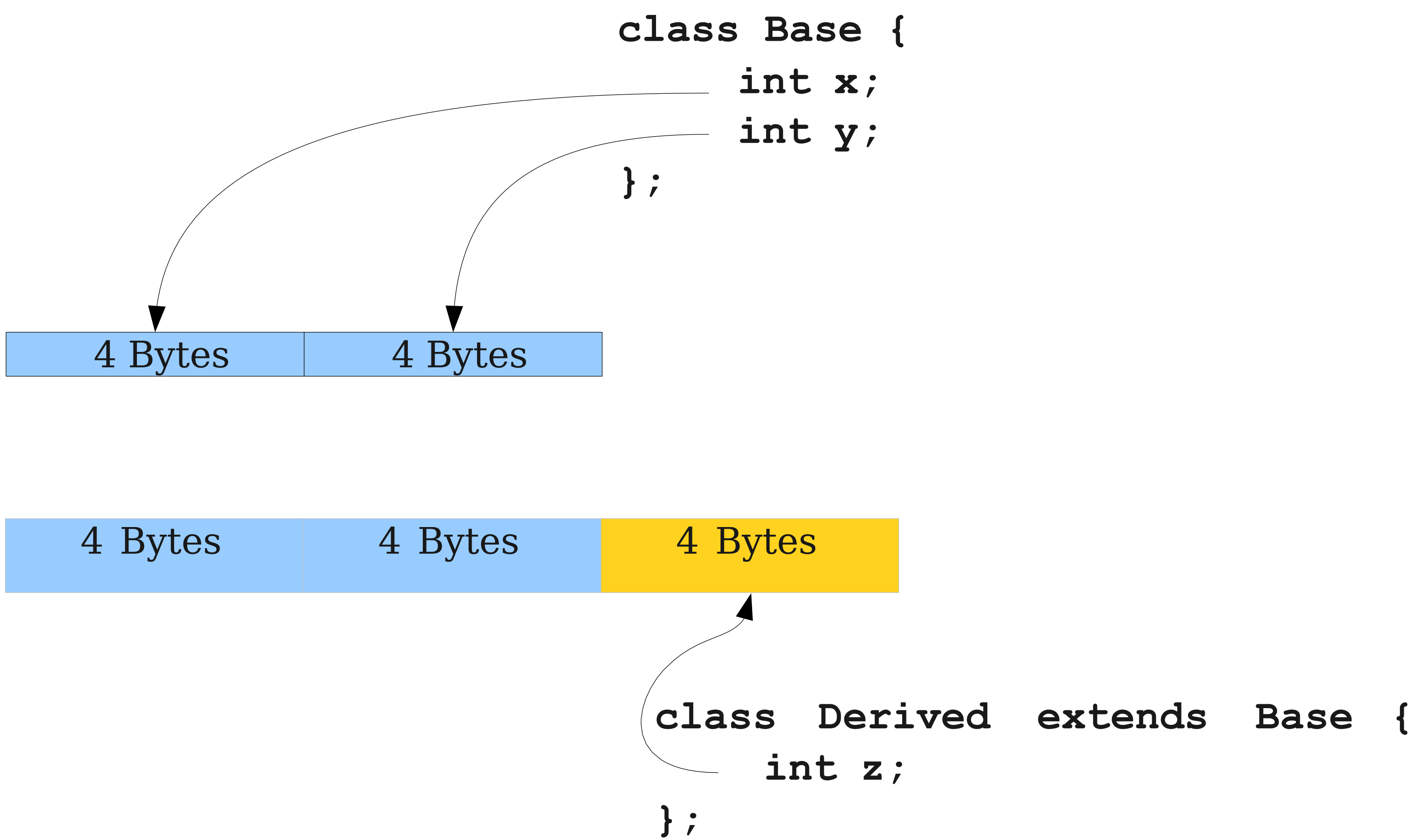
```
struct MyStruct
{   int x;
    char y;
    double z;
};
```



Idea: Keep an internal table inside the compiler containing the offsets of each field.

```
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
```

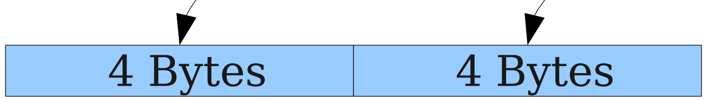
Memory Layouts with Inheritance



Field Lookup With Inheritance

Field Lookup With Inheritance

```
class Base {
    int x;
    int y;
};
```

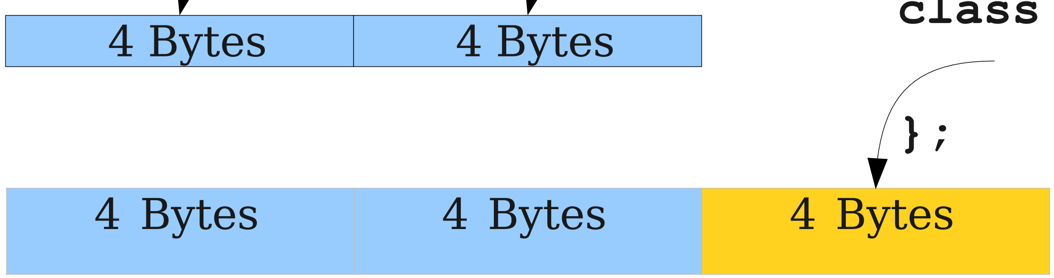



```
class Derived extends Base {
    int z;
};
```

Field Lookup With Inheritance

```
class Base {
    int x;
    int y;
};
```

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




```
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
```

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 Clever

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

MyClass m = new MyClass;
// m.myFunction(137);
MyClass_myFunction(m, 137);
```

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?

Dynamic Dispaching Example

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

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

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

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 initial idea

- It's slow.
  - Number of checks is O(C), where C is the number of classes the dispatch might refer to.
- It's infeasible in most languages.
  - 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("Base");
  }
}
```

sayHi	Base.x
sayHi	Base.x	Derived.y

Code for **Base.sayHi**

Code for **Derived.sayHi**

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

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

More Virtual Function Tables

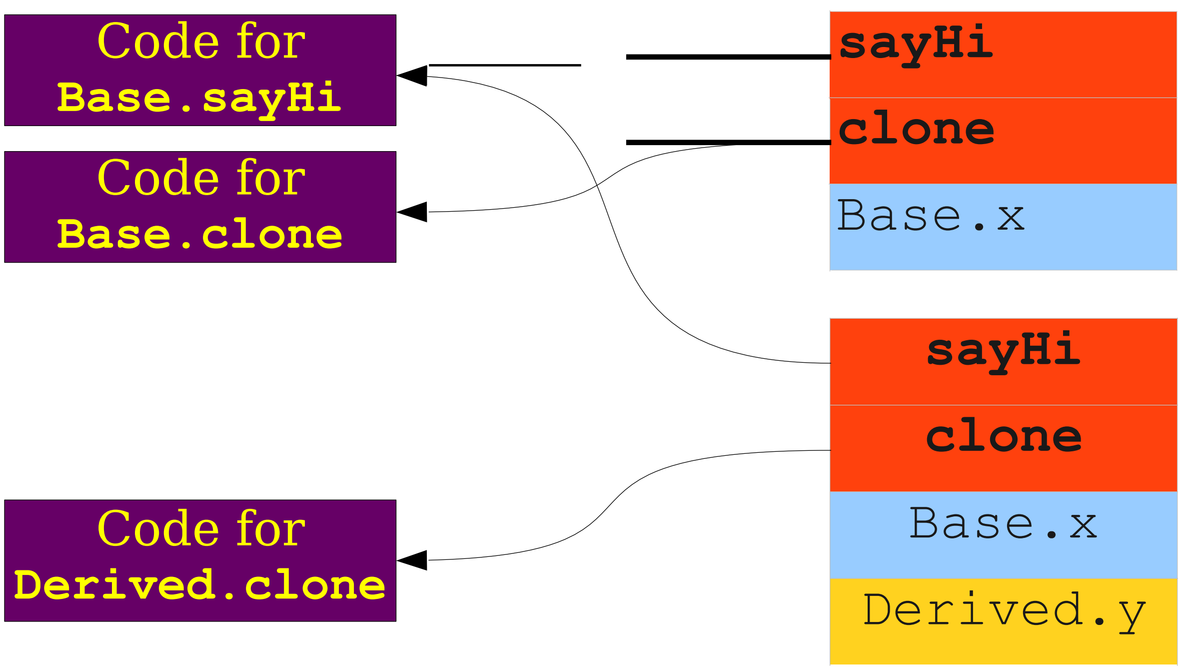
```
class Base {
  int x;
  void sayHi() {
    Print("Hi Mom!");
  }
  Base clone() {
    return new Base;
  }
}
```

```
class Derived extends Base {
  int y;
  Derived clone() {
    return new Derived;
  }
}
```

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;
  }
}
```



Analyzing our Approach

- Advantages:
  - Time to determine function to call is O(1).
  - (and a good O(1) too!)
- What are the disadvantages?

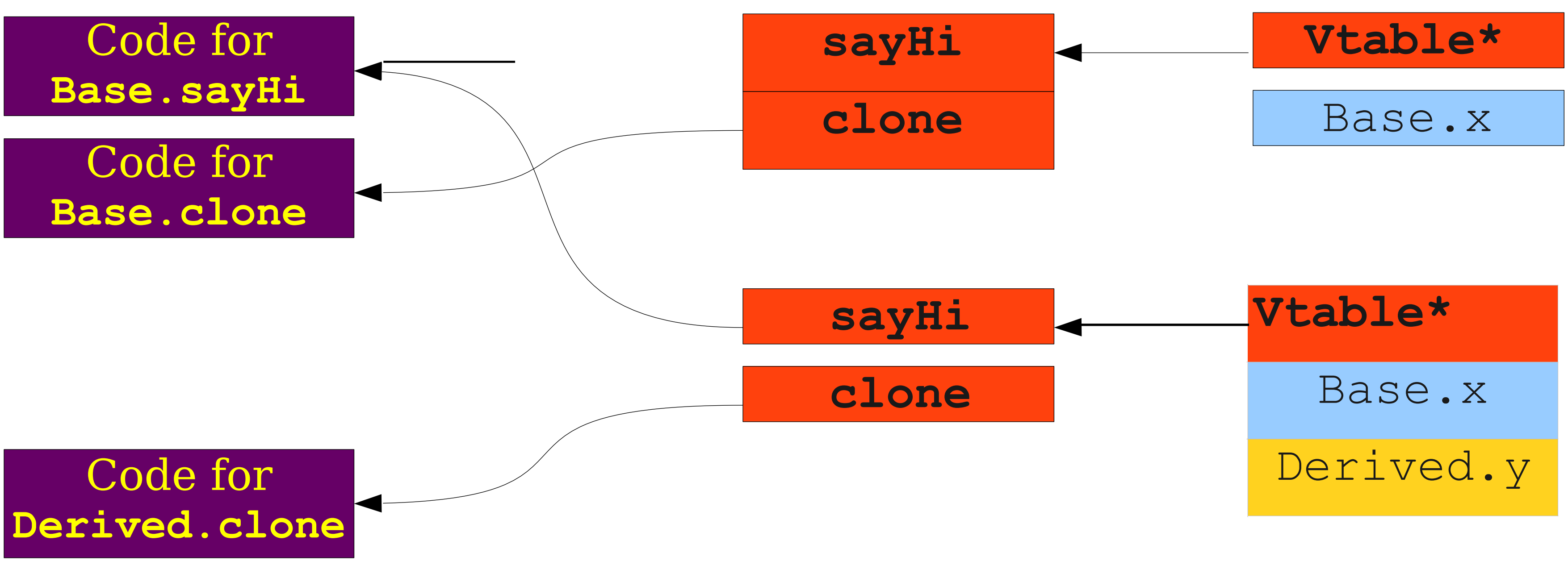
Analyzing our Approach

- 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("Hi Mom!");
  }
  Base clone() {
    return new Base;
  }
}
```

```
class Derived extends Base {
  int y;
  Derived clone() {
    return new Derived;
  }
}
```



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.

Inheritance in PHP

```
class Base {
    public function sayHello() {
        echo "Hi!  I'm Base.";
    }
}

class Derived extends Base {
    public function sayHello() {
        echo "Hi! I'm Derived.";
    }
}

$b = new Base();
$b->sayHello();

$d = new Derived();
$d->sayHello();

$b->missingFunction();

$fnName = "sayHello";
$b->$fnName();
```

We don't know the method name statistically!

```
> Hi!  I'm Base.
  Hi!  I'm Derived.
  ERROR:  Base::missingFunction
          is not defined
  Hi!  I'm Base.
```

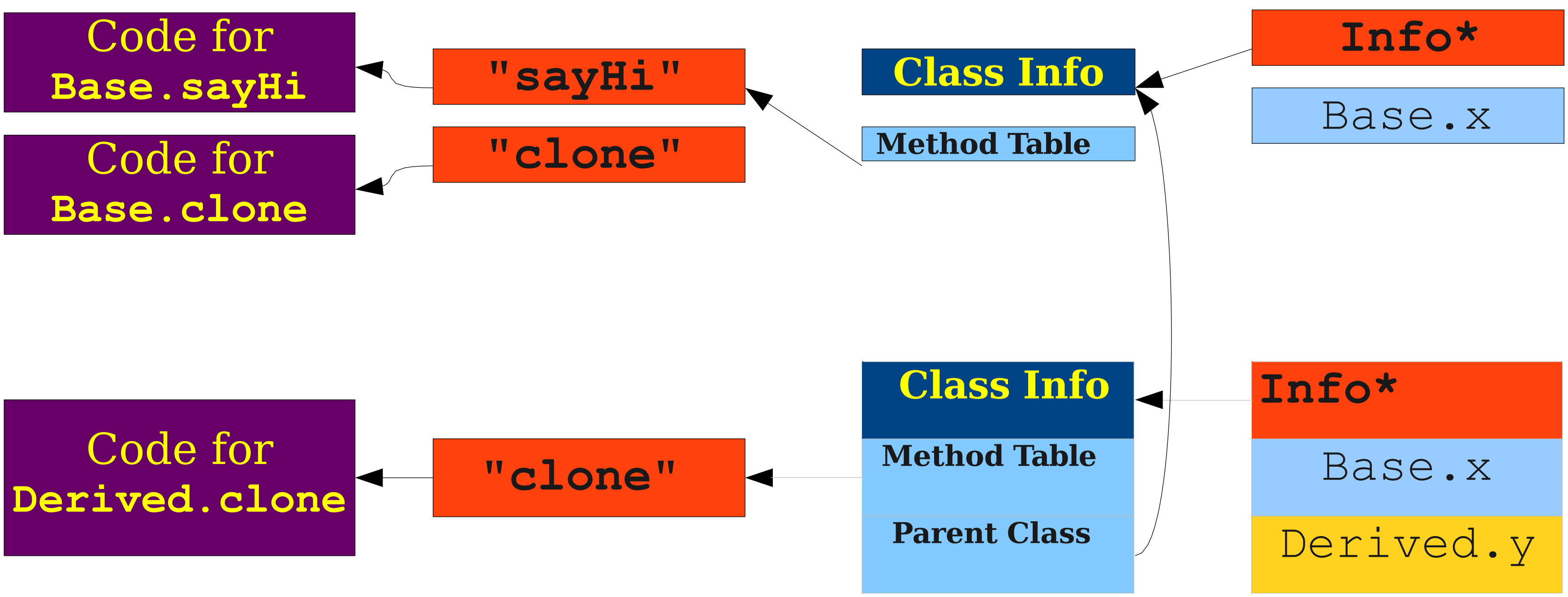
PHP Inhibits Vtables

- **Call-by-string bypasses the vtable optimization**.
  - Impossible to statically determine contents of any string.
  - Would have to determine index into vtable at runtime.
- **No static type information on objects**.
  - Impossible to statically determine whether a given method exists at all.
- **Plus a few others**:
  - `eval` keyword executes arbitrary PHP code; could introduce new classes or methods.

Inheritance without Vtables

```
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;
  }
}
```

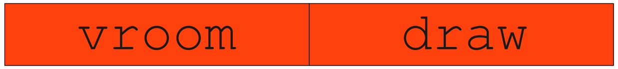


Interfaces, when the problem emerges

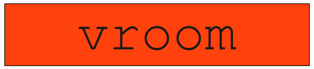
Vtables and Interfaces

```
interface Engine {
    void vroom();
}
interface Visible {
    void draw();
}
class PaintedEngine implements Engine, Visible {
    void vroom() { /* … */ }
    void draw() { /* … */ }
}
class JetEngine implements Engine {
    void vroom() { /* … */ }
}
class Paint implements Visible {
    void draw() { /* … */ }
}

Engine e1  = new PaintedEngine;
Engine e2  = new JetEngine;
e1.vroom();
e2.vroom();
Visible v1 = new PaintedEngine;
Visibie v2 = new Paint;
v1.draw();
v2.draw();
```

PaintedEngine vtable



JetEngine vtable



Vtables and Interfaces

```
interface Engine {
    void vroom();
}
interface Visible {
    void draw();
}
class PaintedEngine implements Engine, Visible {
    void vroom() { /* … */ }
    void draw() { /* … */ }
}
class JetEngine implements Engine {
    void vroom() { /* … */ }
}
class Paint implements Visible {
    void draw() { /* … */ }
}

Engine e1  = new PaintedEngine;
Engine e2  = new JetEngine;
e1.vroom();
e2.vroom();
Visible v1 = new PaintedEngine;
Visibie v2 = new Paint;
v1.draw();
v2.draw();
```

`PaintedEngine` vtable

vroom	draw

`JetEngine` vtable

vroom

?? `Engine` vtable ??

? vroom ?

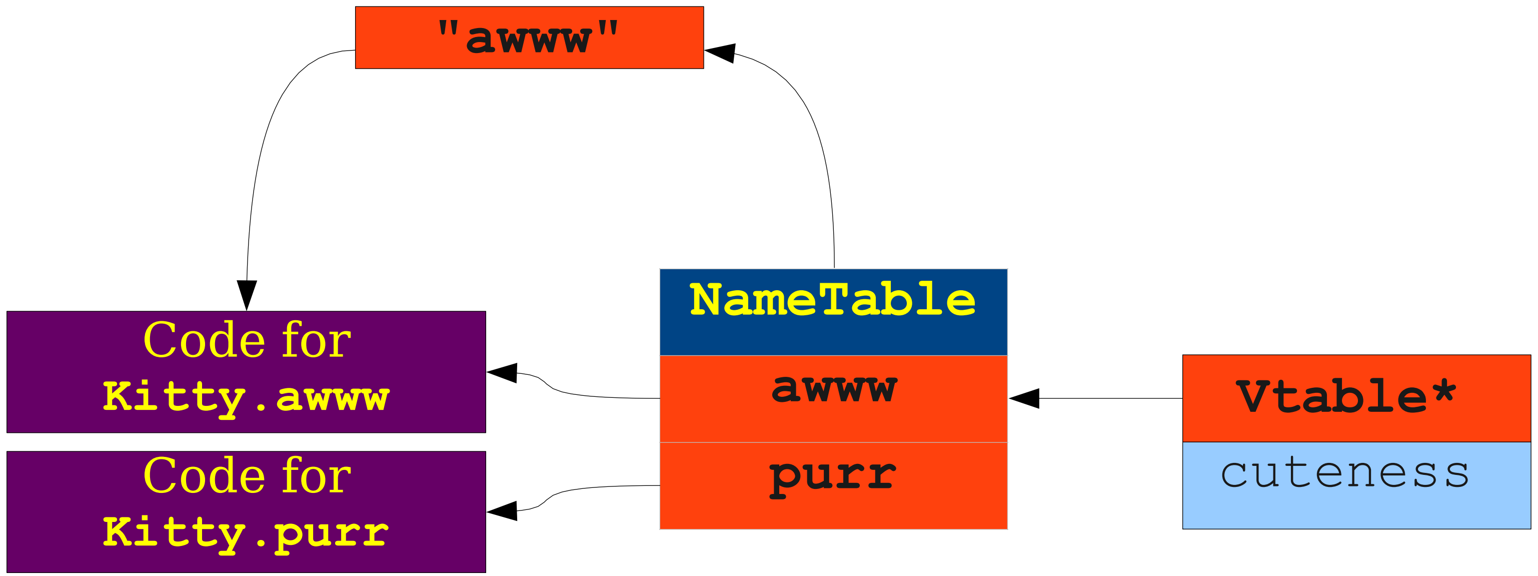
Interfaces via String Lookup

- Idea: A hybrid approach.
- Use vtables for standard (non-interface) dispatch.
- Use the more general, string-based lookup for interfaces.

Object Layout with Interfaces

```
class Kitty implements Adorable {
     int cuteness;
     void awww() {
          Print("Meow");
     }
     void purr() {
          Print("Purr");
     }
}
```

```
interface Adorable {
     void awww();
}
```



Analysis of the Approach

- Dynamic dispatch through object types still O(1).
- Interface dispatches:
  - Search in name table for function name.
- O(Mn), where M is the number of methods and n is the length of the method name.
- O(n) with hash table implementation

Optimizing Interface Dispatch

- Assign a unique number to each interface method.
- Replace hash table of strings with hash table of integers.
- Vtable effectively now a hash table instead of an array.
- Cost to do an interface dispatch now O(1).
  - (But still more expensive than a standard dynamic dispatch.)
- Would this work in PHP?
  - **No**; can still do string-based lookups directly.

Remembering Dispatches

- A particular interface dispatch site often refers to the same method on the majority of its calls.
- **Idea**: Have a global variable for each call site that stores
  - The type of the last object used there, and
  - What method the call resolved to.
- When doing an interface dispatch, check whether the type of the receiver matches the cached type.
  - If so, just use the known method.
  - If not, fall back to the standard string-based dispatch.
- This is called **inline caching**.

Vtables Revisited

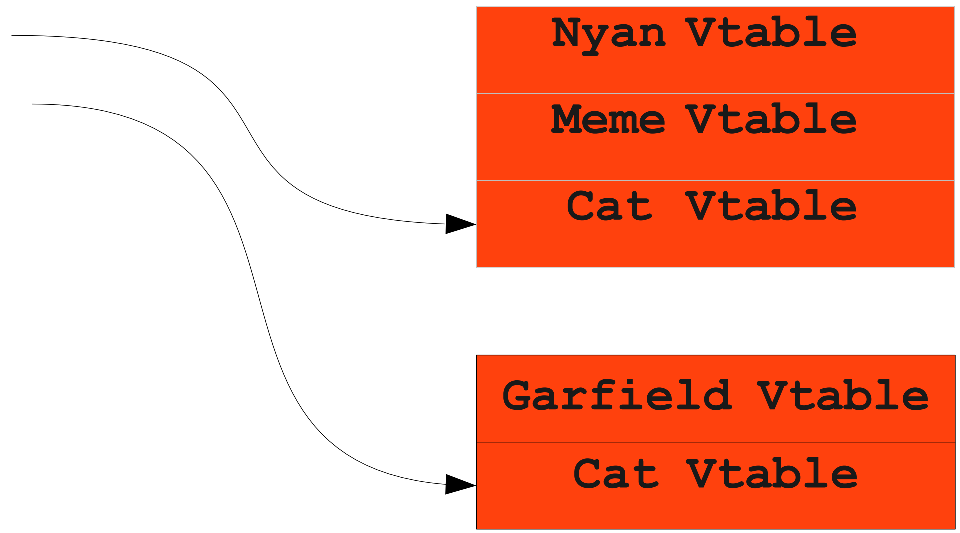
- **Recall**: Why do interfaces complicate vtable layouts?
- **Answer**: Interface methods must have a consistent position in all vtables.
- **Idea**: What if we have multiple vtables per object, one for each interface?
  - Allows interface methods to be positioned independently of one another.
  - Allows for fast vtable lookups relative to string-based approach.
- This is tricky but effective; most C++ implementations use this approach.

A Partial Solution

```
class Nyan implements Meme, Cat {
     /* … */
}

class Garfield implements Cat {
     /* … */
}
```

```
Cat c1 = new Nyan;
Cat c2 = new Garfield;
```

c1.meow();
c2.meow();



Looking in the Wrong Place

```
interface Cat {
    void meow();
}
class Garfield implements Cat {
    int totalSleep;
    void meow() {
        totalSleep --;
        Print("I'm
        tired.");
    }
}

Cat g = new Garfield;
g.meow();
```

Code for Garfield::meow(Garfield* this)

```
Code for Garfield::meow(Garfield* this)
    Look up the integer 8 bytes past 'this'
    Read its value into memory
    Subtract one from the value
    Store the value back into memory
```

Multiple Vtables

```
class A {
public:
  int a;
  virtual void v() {}
};

class B {
public:
  int b;
  virtual void w();
};

class C : public A, public B {
public:
  int c;
  virtual void v() {}
};
```

```
                                     +-----------------------+
                                     |     0 (top_offset)    |
                                     +-----------------------+
c --> +----------+                   | ptr to typeinfo for C |
      |  vtable  |-------->          +-----------------------+
      +----------+                   |        A::v()         |
      |     a    |                   +-----------------------+
      +----------+                   |    -8 (top_offset)    |
      |  vtable  |---+               +-----------------------+
      +----------+   |               | ptr to typeinfo for C |
      |     b    |   +--->           +-----------------------+
      +----------+                   |        B::w()         |
      |     c    |                   +-----------------------+
      +----------+
```

Multiple Vtables

```
class A {
public:
  int a;
  virtual void v() {}
};

class B {
public:
  int b;
  virtual void w();
};

class C : public A, public B {
public:
  int c;
  virtual void v() {}
};

C* c = new C();
B* b = c;
A* a = c;
C* a_c = static_cast<C*>(a);
C* b_c = static_cast<C*>(b);
```

```
                                   +-----------------------+
                                   |     0 (top_offset)    |
                                   +-----------------------+
c --> +----------+                 | ptr to typeinfo for C |
      |  vtable  |--------> +-----------------------+
      +----------+                 |         A::v()        |
      |     a    |                 +-----------------------+
      +----------+                 |    -8 (top_offset)    |
      |  vtable  |---+             +-----------------------+
      +----------+   |             | ptr to typeinfo for C |
      |     b    |   +---> +-----------------------+
      +----------+                 |         B::w()        |
      |     c    |                 +-----------------------+
      +----------+
```

Multiple Vtables

```
class A {
public:
  int a;
  virtual void v() {}
};

class B {
public:
  int b;
  virtual void w();
};

class C : public A, public B {
public:
  int c;
  virtual void v() {}
};

C* c = new C();
B* b = c;
A* a = c;
C* a_c = static_cast<C*>(a);
C* b_c = static_cast<C*>(b);
```

```
                                          +-----------------------+
                                          |     0 (top_offset)    |
                                          +-----------------------+
a --> c --> +----------+                  | ptr to typeinfo for C |
            |  vtable  |--------> +-----------------------+
            +----------+                  |        A::v()         |
            |     a    |                  +-----------------------+
            +----------+                  |    -8 (top_offset)    |
            |  vtable  |---+              +-----------------------+
            +----------+   |              | ptr to typeinfo for C |
            |     b    |   +---> +-----------------------+
            +----------+                  |        B::w()         |
            |     c    |                  +-----------------------+
            +----------+
```

Multiple Vtables

```
class A {
public:
  int a;
  virtual void v() {}
};

class B {
public:
  int b;
  virtual void w();
};

class C : public A, public B {
public:
  int c;
  virtual void v() {}
};

C* c = new C();
B* b = c;
A* a = c;
C* a_c = static_cast<C*>(a);
C* b_c = static_cast<C*>(b);
```

```
                                        +-----------------------+
                                        |    0 (top_offset)     |
                                        +-----------------------+
a --> c --> +----------+                | ptr to typeinfo for C |
            |  vtable  |--------->      +-----------------------+
            +----------+                |        A::v()         |
            |    a     |                +-----------------------+
      b --> +----------+                |   -8 (top_offset)     |
            |  vtable  |---+            +-----------------------+
            +----------+   |            | ptr to typeinfo for C |
            |    b     |   +--->        +-----------------------+
            +----------+                |        B::w()         |
            |    c     |                +-----------------------+
            +----------+
```

Multiple Vtables

```
class A {
public:
  int a;
  virtual void v() {}
};

class B {
public:
  int b;
  virtual void w();
};

class C : public A, public B {
public:
  int c;
  virtual void v() {}
};

C* c = new C();
B* b = c;
A* a = c;
C* a_c = static_cast<C*>(a);
C* b_c = static_cast<C*>(b);
```

```
                                     +-----------------------+
                                     |     0 (top_offset)    |
                                     +-----------------------+
a --> c --> +----------+             | ptr to typeinfo for C |
            |  vtable  |-------->    +-----------------------+
            +----------+             |        A::v()         |
            |    a     |             +-----------------------+
      b --> +----------+             |    -8 (top_offset)    |
            |  vtable  |---+         +-----------------------+
            +----------+   |         | ptr to typeinfo for C |
            |    b     |   +--->     +-----------------------+
            +----------+             |        B::w()         |
            |    c     |             +-----------------------+
            +----------+
```

b_c = c - 8 bytes;

Multiple Vtables

```
class A {
public:
  int a;
  virtual void v() {}
};

class B {
public:
  int b;
  virtual void w();
};

class C : public A, public B {
public:
  int c;
  virtual void v() {}
};

C* c = new C();
B* b = c;
A* a = c;
C* a_c = static_cast<C*>(a);
C* b_c = static_cast<C*>(b);
```

```
                                   +-----------------------+
                                   |     0 (top_offset)    |
                                   +-----------------------+
a --> c --> +----------+           | ptr to typeinfo for C |
            |  vtable  |--------> +-----------------------+
            +----------+           |        A::v()         |
            |    a     |           +-----------------------+
      b --> +----------+           |    -8 (top_offset)    |
            |  vtable  |---+       +-----------------------+
            +----------+   |       | ptr to typeinfo for C |
            |    b     |   +---> +-----------------------+
            +----------+           |        B::w()         |
            |    c     |           +-----------------------+
            +----------+
```

b_c = c - 8 bytes;

For multiple inheritance (or interface):

1) Keeping one place for each data (MI)
2) Down-cast, how to know dynamic type?

Analysis of Vtable Deltas

- Cost to invoke a method is O(1) regardless of the number of interfaces.
- Also a **fast** O(1); typically much better than a hash table lookup.
- Size of an object increases by O(`I`), where `I` is the number of interfaces.
- Cost to create an object is O(`I`), where `I` is the number of interfaces.
  - (Why?)

Comparison of Approaches

- String-based lookups have small objects and fast object creation but slow dispatch times.
  - Only need to set one vtable pointer in the generated object.
  - Dispatches require some type of string comparisons.
- Vtable-based lookups have larger objects and slower object creation but faster dispatch times.
  - Need to set multiple vtable pointers in the generated object.
  - Dispatches can be done using simple arithmetic.

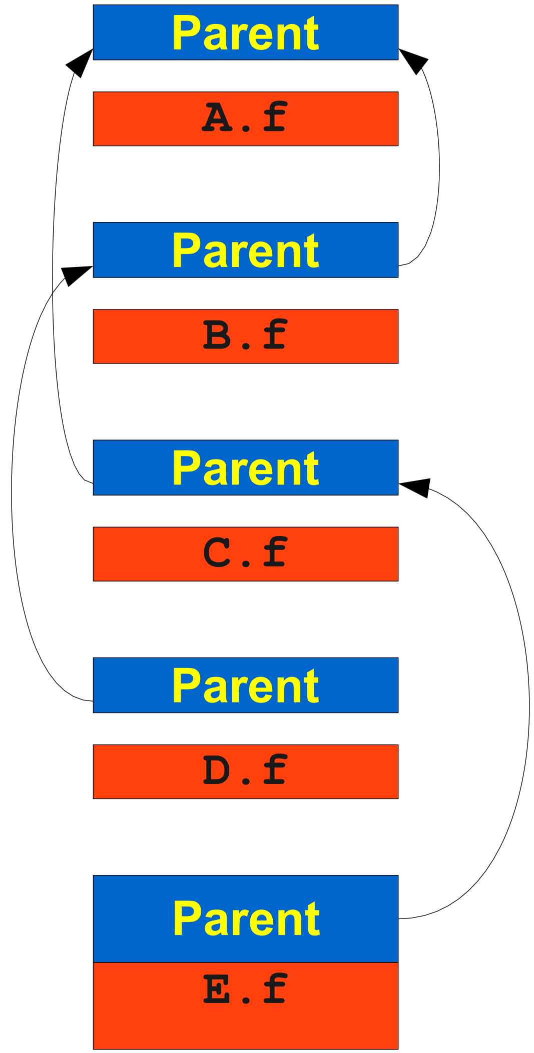
Implementing Dynamic Type Checks

Dynamic Type Checks

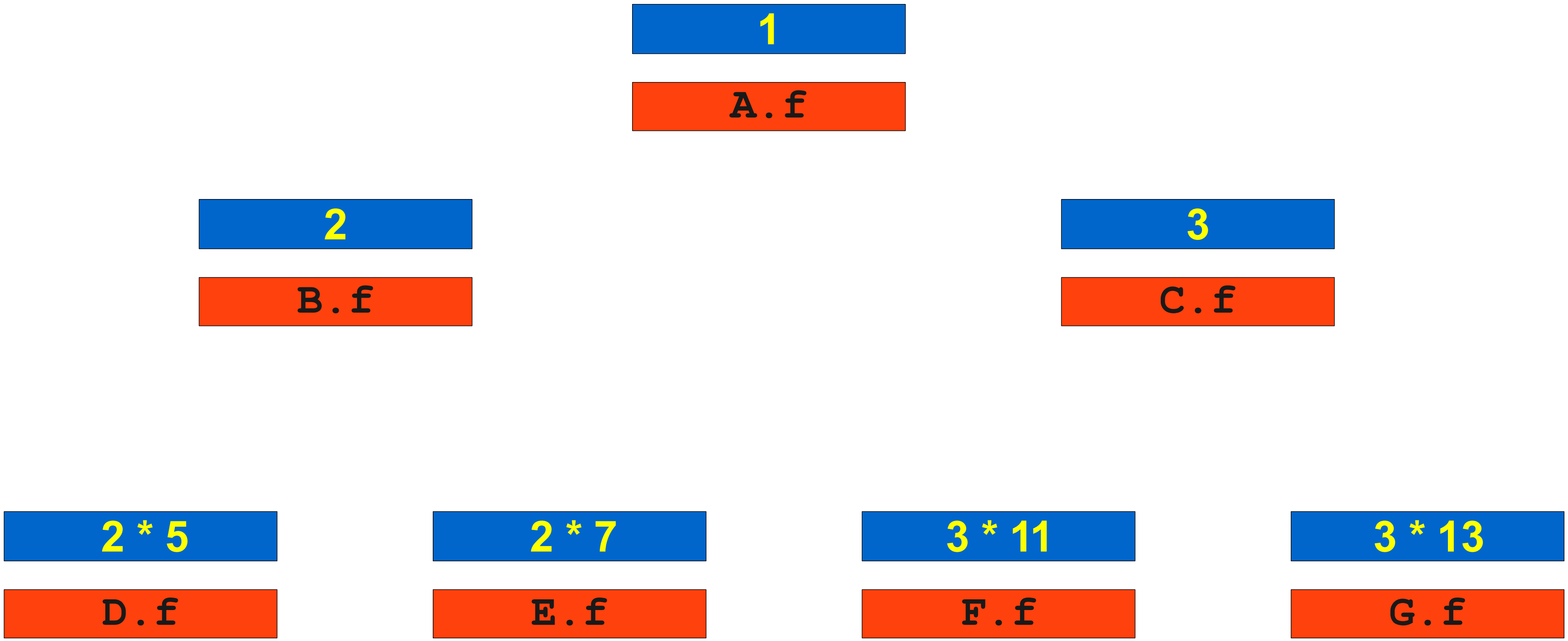
- Many languages require some sort of dynamic type checking.
  - Java's **`instanceof`**, C++'s **`dynamic_cast`**, any dynamically-typed language.
- May want to determine whether the dynamic type is *convertible* to some other type, not whether the type is *equal*.
- How can we implement this?

A Pretty Good Approach

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

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

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

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

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



A Marvelous Idea

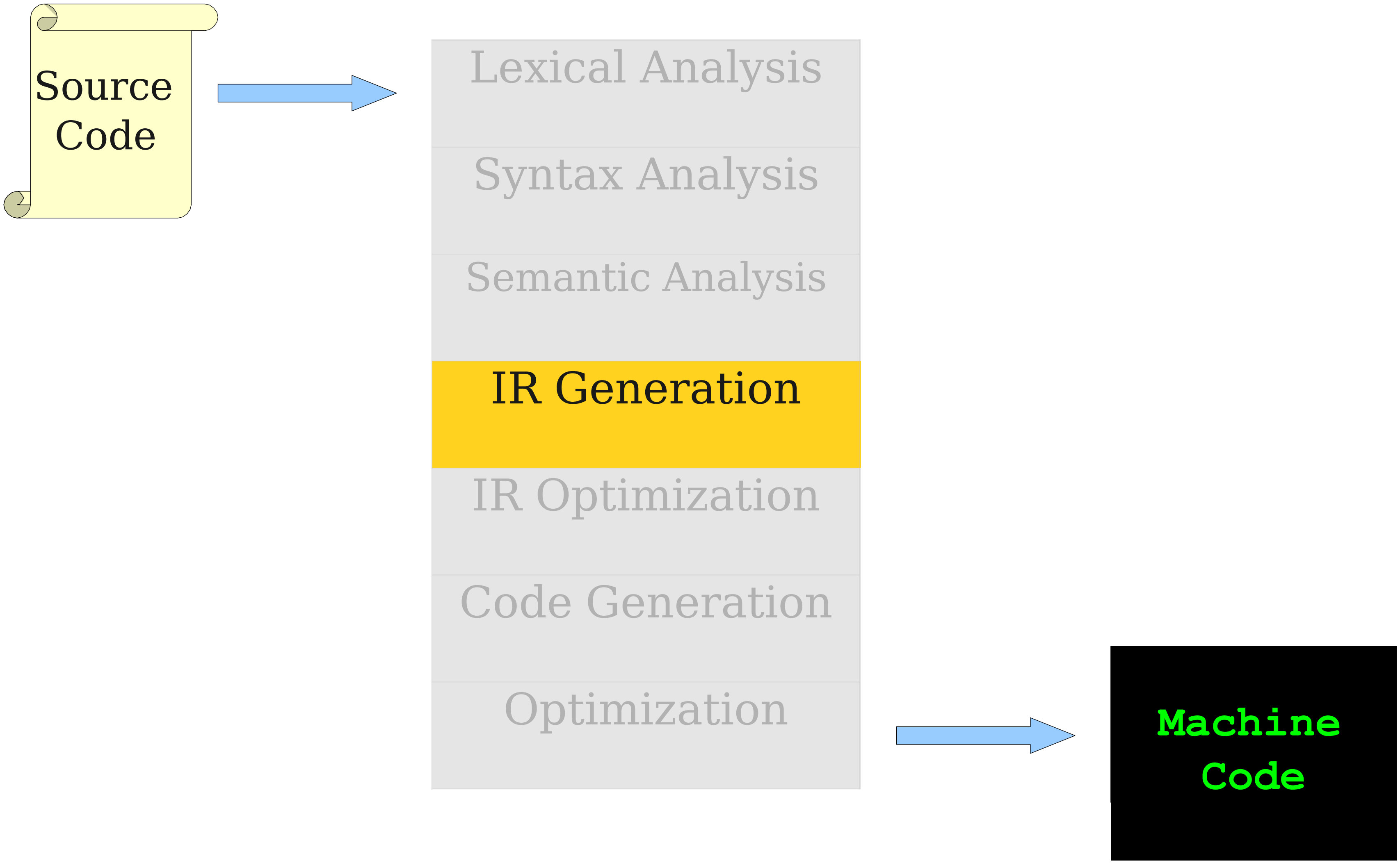


Next Time

- **Three-Address Code IR.**
- **IR Generation.**

Three-Address Code IR

Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

TAC

- TAC for expressions
- TAC for function call
- TAC for objects

Generating TAC

TAC commands

- var1 = [constant | var2];
- var1 = [constant | var2] op [constant | var3];
- *(var1 [+ constant]) = var2
- var1 = *(var2 [+ constant])

Sample TAC Code

```
int  a;
int  b;
int  c;
int d;

a = b + c + d;
b = a * a + b * b;
```

```
_t0 = b + c;
a = _t0 + d;
_t1 = a * a;
_t2 = b * b;
b = _t1 + _t2;
```

cgen for Basic Expressions

cgen for Basic Expressions

cgen(k) = { // k is a constant
    Choose a new temporary t
    Emit($t = k$);
    Return t
}

cgen for Basic Expressions

cgen(k) = { // k is a constant
    Choose a new temporary t
    Emit($t = k$);
    Return t
}

cgen(id) = { // id is an identifier
    Choose a new temporary t
    Emit($t = id$)
    Return t
}

cgen for Binary Operators

cgen for Binary Operators

cgen(e_1 + e_2) = {

Choose a new temporary t

Let t_1 = **cgen**(e_1)

Let t_2 = **cgen**(e_2)

Emit($t = t_1 + t_2$)

Return t

}

cgen for Simple Statements

TAC commands

- var1 = [constant | var2];
- var1 = [constant | var2] op [constant | var3];
- *(var1 [+ constant]) = var2
- var1 = *(var2 [+ constant])

- Labels,
  - Goto label
  - IfZ var Goto label

cgen for `while` loops

cgen(`while` (*expr*) *stmt*) = {

Let L_{before} be a new label.

Let L_{after} be a new label.

Emit(L_{before}:)

Let t = **cgen**(*expr*)

Emit(`IfZ` t `Goto` L_{after})

cgen(*stmt*)

Emit(`Goto` L_{before})

Emit(L_{after}:)

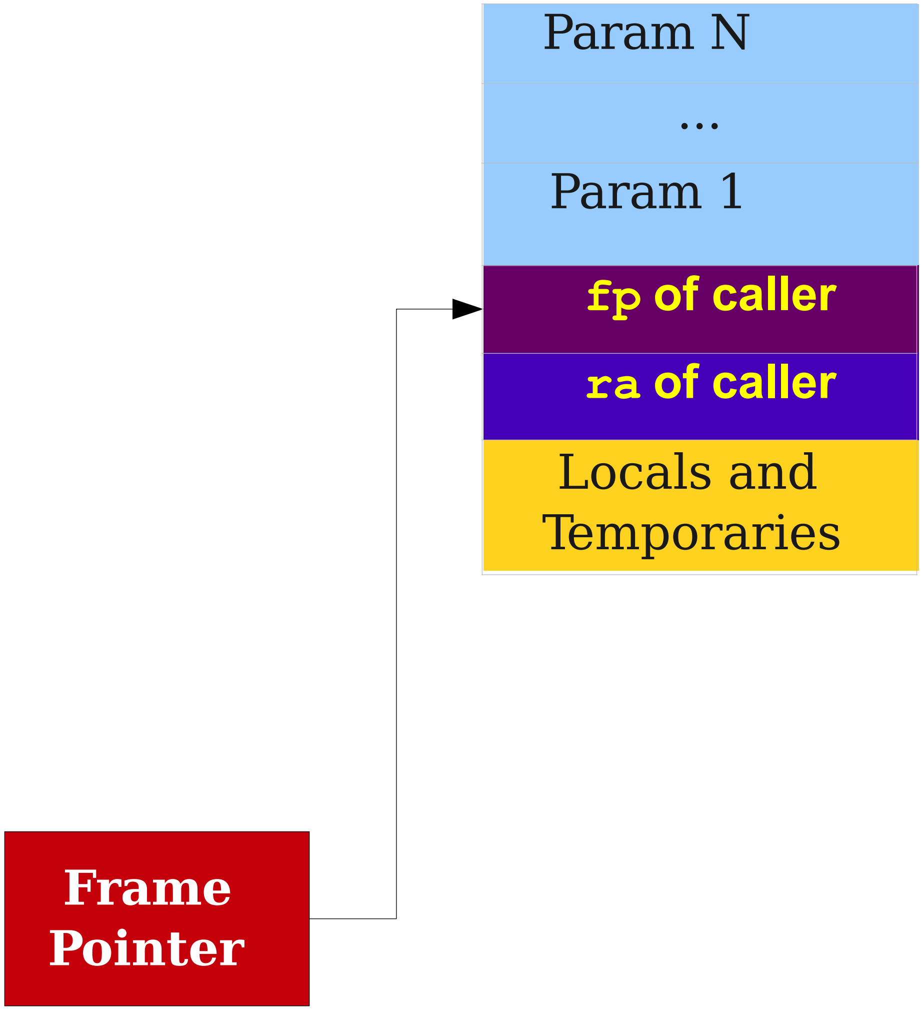
}

Compiling Functions

- BeginFunc N
  - Reserves N bytes
- EndFunction
  - Frees N bytes

- Call

Physical Stack Frames



A Complete Decaf Program

```
void main() {
    int x, y;
    int m2 = x * x + y * y;

    while (m2 > 5) {
        m2 = m2 - x;
    }
}
```

```
main:
    BeginFunc 24;
    _t0 = x * x;
    _t1 = y * y;
    m2 = _t0 + _t1;
_L0:
    _t2 = 5 < m2;
    IfZ _t2 Goto _L1;
    m2 = m2 - x;
    Goto _L0;
_L1:
    EndFunc;
```

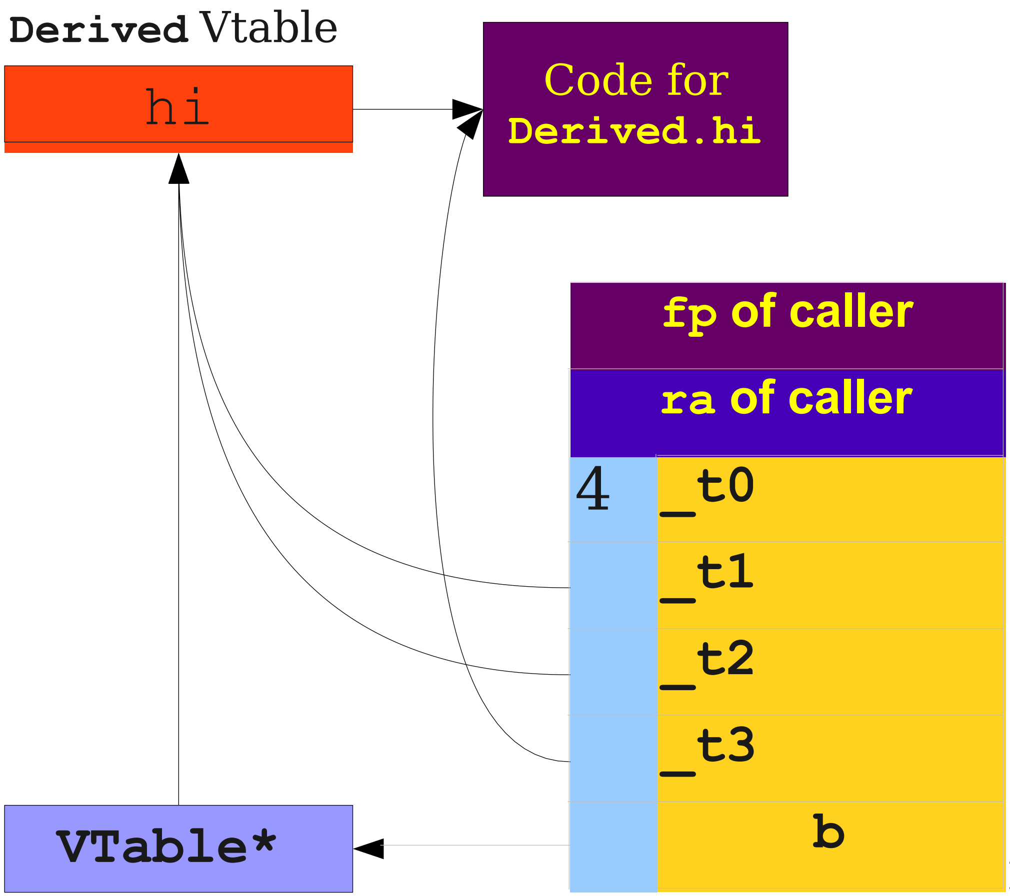
A Complete Decaf Program

```
void f(int a) {
}

void main() {
   int x, y;
   f(x);
}
```

```
f:
   BeginFunc 0;

   EndFunc;

main:
   BeginFunc 8;
   PushParam x;
   Call f;
   PopParams 4;
   EndFunc;
```

Objects

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

Load Function

Next Time

- **Intro to IR Optimization**
  - Basic Blocks
  - Control-Flow Graphs
  - Local Optimizations