CSE 12:
Basic data structures and object-oriented design

Jacob Whitehill
jake@mplab.ucsd.edu

Lecture Six
9 Aug 2011

Type-safety and casting.

Type-safety in Java

- As mentioned in Lecture Three, Java was designed from the ground up to offer *security*.
- One aspect of security is ensuring that a variable that, for example, is supposed to point to a **`String`** doesn't actually point to an **`Integer`**.

```
// Won't compile
final String s = new Integer(6);
```

- This form of security is known as *type-safety*.

Type-safety in Java

- That example was somewhat obvious; let's look at a more subtle one...

```
final Object o = new Integer();
final String s = (String) o;
```

This code will compile ok, ...

Type-safety in Java

- That example was somewhat obvious; let's look at a more subtle one...

```
final Object o = new Integer();
final String s = (String) o;  // Compiles ok
```

...but at run-time, the second statement will trigger a **`ClassCastException`** -- an **`Integer`** is never also a **`String`**!

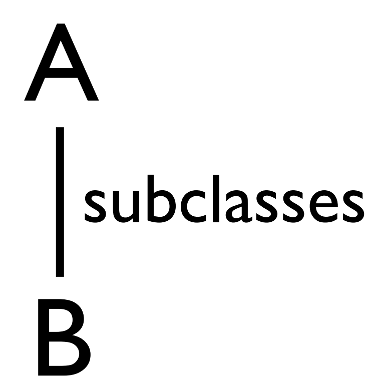
Type-safety

- Java and the JVM enforce type-safety:
  - Every `Object` knows what kind of class it is, what its parent class is, and all the interfaces that it implements.
  - If you attempt to "cast" an object into a type with which it is not compatible, then this will trigger a `ClassCastException`.
    - Your program will terminate.

Casting

- In object-oriented languages like Java, objects are *cast* into different classes/interfaces when we assign them to reference variables of *different types*.

- Consider:




```
class A { ...
}
class B extends A { ...
}

B b = new B();
A a = b;         // Upcast from B to A.
B b2 = (B) a;    // Downcast from A to B.
```

The terms upcast and downcast have to do with the class hierarchy, in which parent classes are "above" child classes.

Upcasting

```
class A { ...
}
class B extends A { ...
}
```

- If class **B** is a subclass of **A**, and we convert a reference of type **B** to a reference of type **A**, then we are upcasting, e.g.: `A a = b;`
  - Since all objects of type **B** are implicitly also of type **A**, this cast is *guaranteed to succeed.*
  - Every object of type **B** can also be treated as an object of type **A**.
    - All methods and instance variables of **A** are guaranteed to be accessible.

Downcasting

```
class A { ...
}
class B extends A { ...
}
```

- If class **`B`** is a subclass of **`A`**, and we convert a reference of type **`A`** to a reference of type **`B`**, then we are downcasting, e.g.: **`B b = (B) a;`**
  - Since objects of type **`A`** are *not guaranteed* to always also be of type **`B`**, we must explicitly inform the compiler that we "know" that **`b`** is of class **`B`**.
    - We must explicitly "request" the cast by writing **`(B)`**.

Downcasting

- At run-time, before performing the cast from class **`A`** to **`B`**, the JVM will check whether **`b`** is actually a **`B`** object.
  - If it is, then execution proceeds merrily.
  - If not, then the JVM will throw a **`ClassCastException`**.

Example 1

```
class Animal { ...
}

class Snake extends Animal { ...
}

class Plant { ...
}

class Test1 {
  public static void main (String[] args)
  {
    Animal animal = new Animal();
    Snake snake = animal;
  }
}
```

Example 1

```
Jacobs-MacBook-Pro:tmp jake$ javac Test1.java
Test1.java:4: incompatible types
found    : Animal
required: Snake
     Snake snake = animal;
                   ^
1 error
```

Compilation error

Example 2

```
class Animal { ...
}

class Snake extends Animal { ...
}

class Plant { ...
}

class Test2 {
  public static void main (String[] args)
  {
    Animal animal = new Animal();
    Snake snake = (Snake) animal;
  }
}
```

Here we attempt to **downcast** from `Animal` to `Snake`. We "promise" the compiler that `animal` is really a `Snake`.

Example 2

Compiles ok

```
Jacobs-MacBook-Pro:tmp jake$ javac Test2.java

Jacobs-MacBook-Pro:tmp jake$ java Test2
Exception in thread "main"
java.lang.ClassCastException: Animal cannot be cast to Snake
   at Test2.main(Test2.java:4)
```

Run-time error -- our "promise" to the compiler was incorrect.

Example 3

```
class Animal { ...
}

class Snake extends Animal { ...
}

class Plant { ...
}

class Test3 {
  public static void main (String[] args)
  {
    Snake snake = new Snake();
    Animal animal = snake;
  }
}
```

Upcast from `Snake` to `Animal` -- this can never fail because `Snake` subclasses `Animal`. Hence, no "promise" is required.

Example 3

```
Jacobs-MacBook-Pro:tmp jake$ javac Test3.java
Jacobs-MacBook-Pro:tmp jake$ java Test3
Jacobs-MacBook-Pro:tmp jake$
```

Compiles ok

Runs ok

Casting example

- Assume the following class definitions:
```
class Animal { ... }
class Snake extends Animal { ... }
class Plant { ... }
```

- Which of the following cause compiler errors?
  - ```
    Animal animal = new Animal();
    Snake snake = (Snake) animal;
    ```
  - ```
    Animal animal = new Snake();
    Snake snake = (Snake) animal;
    ```
  - ```
    Snake snake = new Snake();
    Animal animal = snake;
    ```
  - ```
    Animal animal = new Snake();
    ```
  - ```
    Snake snake = new Animal();
    ```
  - ```
    Snake snake = new Snake();
    Animal animal = (Animal) snake;
    ```
  - ```
    Plant plant = new Snake();
    Animal animal = (Snake) plant;
    ```
    ```

Casting example

- Assume the following class definitions:

```
class Animal { ... }
class Snake extends Animal { ... }
class Plant { ... }
```

- Which of the following cause compiler errors?

  - ```
    Animal animal = new Animal();
    Snake snake = (Snake) animal;
    ```
  - ```
    Animal animal = new Snake();
    Snake snake = (Snake) animal;
    ```
  - ```
    Snake snake = new Snake();
    Animal animal = snake;
    ```
  - ```
    Animal animal = new Snake();
    ```
  - ```
    Snake snake = new Animal();
    ```
  - ```
    Snake snake = new Snake();
    Animal animal = (Animal) snake;
    ```
  - ```
    Plant plant = new Snake();
    Animal animal = (Snake) plant;
    ```
    ```

Casting example

- Assume the following class definitions:

```
class Animal { ... }
class Snake extends Animal { ... }
class Plant { ... }
```

- Which of the following cause runtime errors?
  - ```
    Animal animal = new Animal();
    Snake snake = (Snake) animal;
    ```
  - ```
    Animal animal = new Snake();
    Snake snake = (Snake) animal;
    ```
  - ```
    Snake snake = new Snake();
    Animal animal = snake;
    ```
  - ```
    Animal animal = new Snake();
    ```
  - ```
    Snake snake = new Animal();
    ```
  - ```
    Snake snake = new Snake();
    Animal animal = (Animal) snake;
    ```
  - ```
    Plant plant = new Snake();
    Animal animal = (Snake) plant;
    ```
    ```

Casting example

- Assume the following class definitions:

```
class Animal { ... }
class Snake extends Animal { ... }
class Plant { ... }
```

- Which of the following cause runtime errors?

  - ```
    Animal animal = new Animal();
    Snake snake = (Snake) animal;
    ```
  - ```
    Animal animal = new Snake();
    Snake snake = (Snake) animal;
    ```
  - ```
    Snake snake = new Snake();
    Animal animal = snake;
    ```
  - ```
    Animal animal = new Snake();
    ```
  - ```
    Snake snake = new Animal();
    ```
  - ```
    Snake snake = new Snake();
    Animal animal = (Animal) snake;
    ```
  - ```
    Plant plant = new Snake();
    Animal animal = (Snake) plant;
    ```
    ```

Casting to interfaces

- We can also cast to an interface type, e.g.:

```
Object o = new DoublyPlant12();
Iterable iterable = (Iterable) o;
```

- Since not every object of `Object` class is guaranteed to implement the `Iterable` interface, we must "downcast" to `Iterable`.
- At run-time, the JVM will check whether `o` is of some class that implements `Iterable`, and throw a `ClassCastException` if it is not.

Importance of type-safety

- Not all languages (e.g., C++) are type-safe.
- Imagine what would happen if the JVM didn't catch the class-cast error in the following code:

Here, we "force" the compiler to treat the **`Integer`** pointer as a **`Student`** pointer.

```
Integer integer = new Integer(123);
Student student = (Student) integer;
student._age = 23;
```

Here we attempt to modify the **`_age`** instance variable of a "**`Student`**" object. But **`student`** actually points to an **`Integer`** object!

Danger in casting

- The outcome of this program can't be good -- we're trying to modify the "`_age`" of an `Integer` object!
- What's going on here in terms of memory?
- Let's first convert this example to Java...

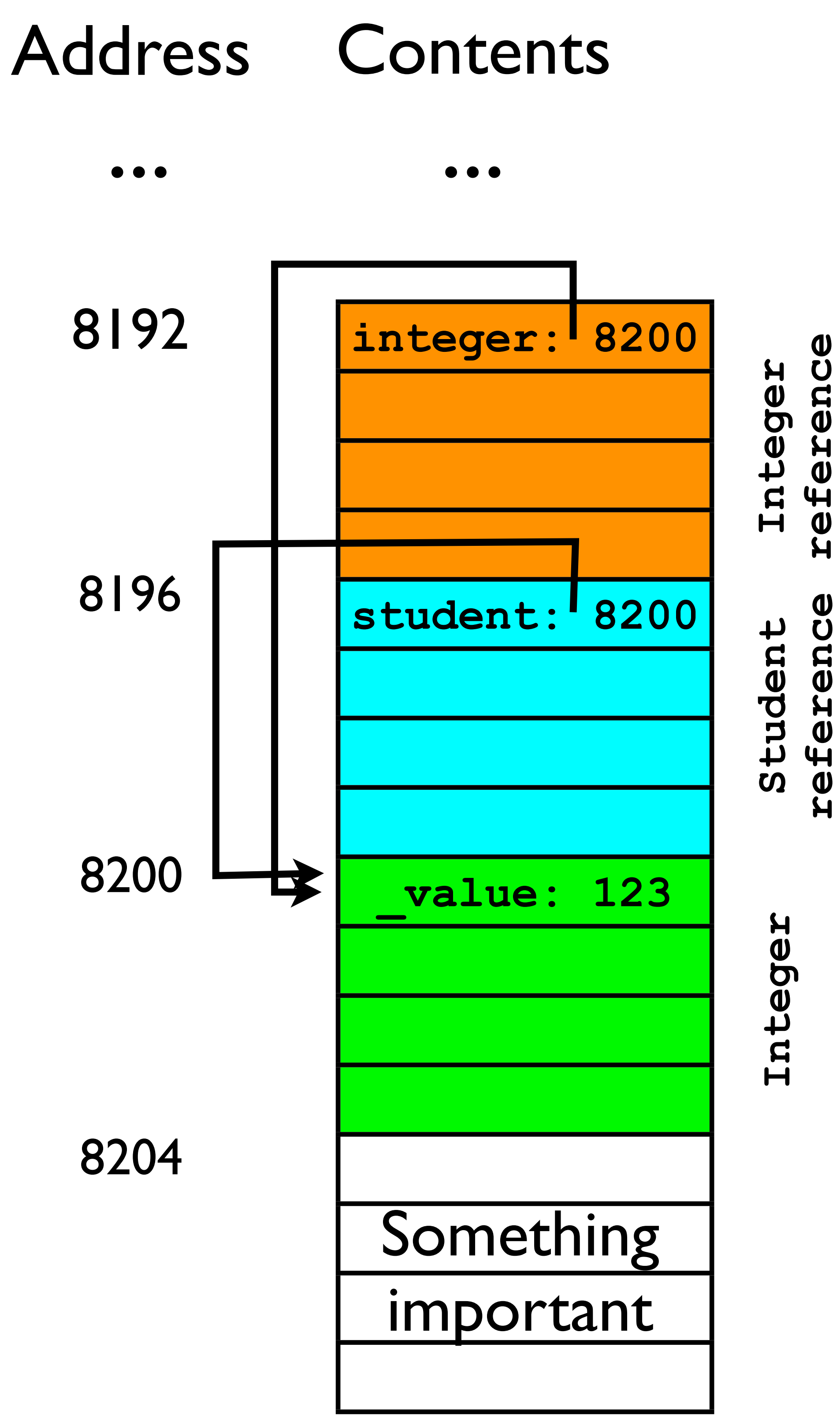
Danger in casting

- Let's assume the following class definitions:

```
class Integer {  // 4 bytes total
  int _value;
}

class Student {  // 8 bytes total
  String _name;
  int _age;
}
```

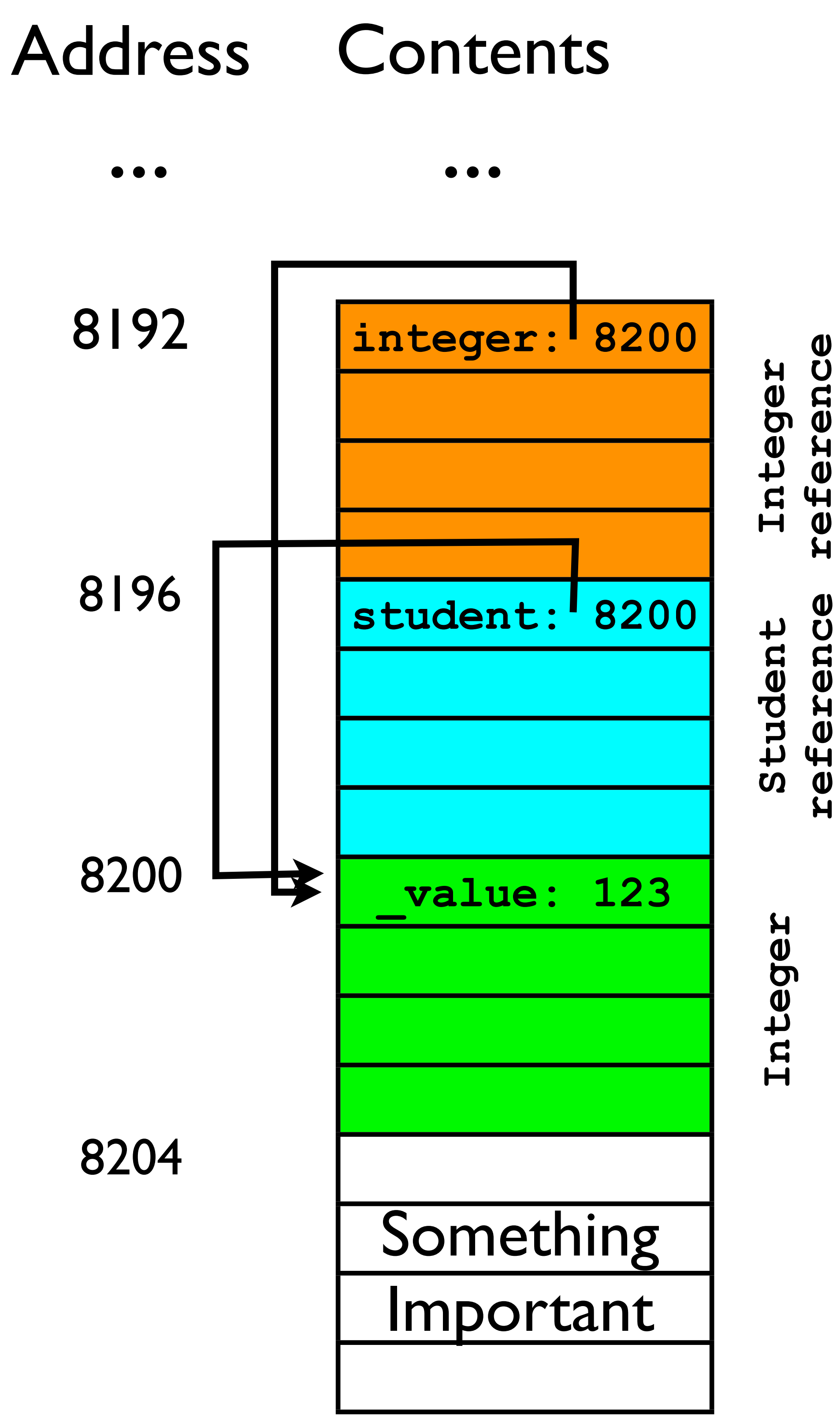
Danger in casting



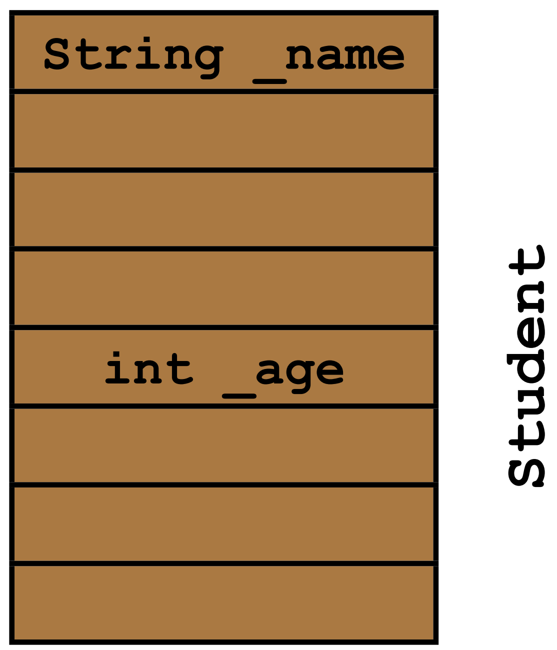

```
Integer integer = new Integer(123);
Student student = (Student) integer;
```

Danger in casting

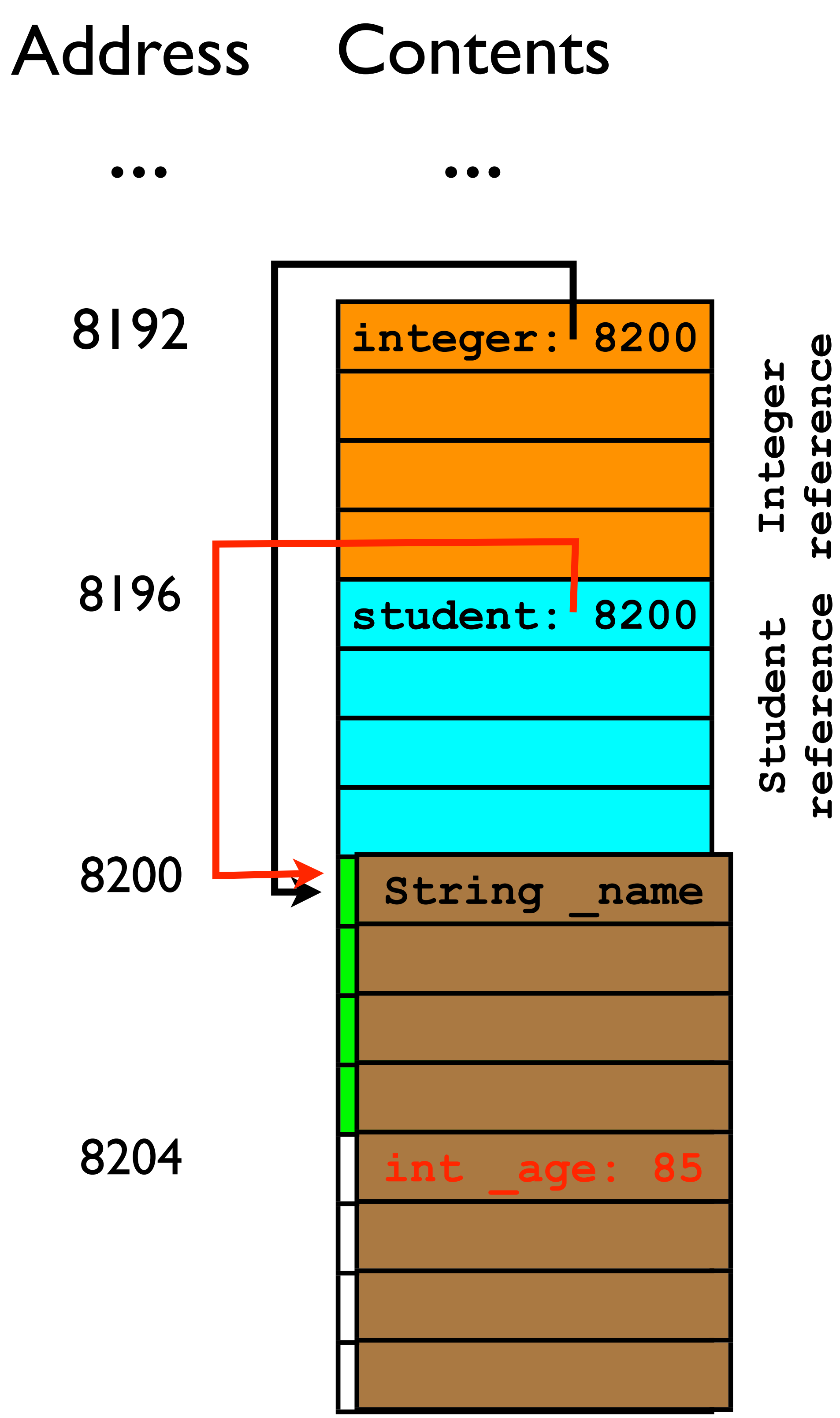
```
Integer integer = new Integer(123);
Student student = (Student) integer;
```



Let's also suppose a "real" `Student` object would look like this:



Danger in casting




```
Integer integer = new Integer(123);
Student student = (Student) integer;
student._age = 85;
```

- In the last line of code, the program attempts to modify the "`_age`" instance variable of the "`Student`" object pointed to by `student`.
  - `_age` *would be* stored at 8204.
- In reality, `student` actually points to an `Integer` object.
  - That `Integer` object does not own address 8204!
  - Something Important has been clobbered.

Clobbering memory

- When you write data outside of a variable's "proper bounds", you are "clobbering memory".
- In the previous example, treating the **`Integer`** like a **`Student`** caused the statement **`student._age = 85`** to overwrite Something Important.
- Without Java's protective type safety, this could cause your program to:
  - Crash.
  - Behave in unexpected ways at some indeterminate point in the future. <== Often worse than crashing.

Clobbering memory

- In some settings (e.g., a web server application that processes data sent from user), treating a variable as an object of the wrong type could be exploited by an attacker.
  - By causing your code to "clobber" the right memory, an attacker might gain control of your entire machine. :-(

Type-safety in Java

- Java and the JVM help to prevent such attacks.
- All downcasts are checked by the JVM to make sure they are valid before execution proceeds.
- As always, this added security comes at a cost:
  - A downcast in Java is slower than a downcast in C++.

Java collections before generics.

Java Collections Framework

- Since Java version 1.2, the JDK has offered pre-built "collections" of various types as part of the Java Collections Framework (JCF).
- The JCF includes such classes as:
  - **`ArrayList`**
  - **`Vector`**
  - **`HashTree`**
  - **`Set`**
  - etc., etc.

CSE12 Collections

- In this course, we have worked on two "collections" -- `ArrayList`, and `DoublyLinkedList12`.
- Similar to the JCF collections in Java 1.2, our collections have dealt with Objects:
  - `public void add (Object o);`
  - `public Object get (int index);`
- Every object in Java is of type `Object`; hence, these collections can store variables of *any type*.

Collections of Objects

- Hence, the same class **`ArrayList`** can be used to create a list of **`Strings`** as well as a list of **`Integers`**:

```
final ArrayList listOfStrings = new ArrayList();
listOfStrings.add("yo");

final ArrayList listOfIntegers = new ArrayList();
listOfIntegers.add(new Integer(32));
```

- This is convenient -- we don't have to create a two different classes to store **`Strings`** versus **`Integers`**.

Downside of downcasting

- Unfortunately, the fact that the `List12` interface takes and returns `Objects` also means that we have to *downcast* the `Object` every time we call `get(index)`:

```
listOfStrings.add("hello");
...
final String s = (String) listOfStrings.get(0);
```

- Having to downcast every time is both *tedious* and *distracting* because it litters the code with parentheses and class names.

Downside of downcasting

- There's also a security reason why downcasting an `Object` returned by a collection is bad:
  - We may accidentally try to downcast an `Object` to an incompatible type.

Downside of downcasting

- Consider a method in which you use several collections to store data of several types:

```
ArrayList list1, list2, list3;
list1 = new ArrayList();  // for Strings
list2 = new ArrayList();  // for Integers
list3 = new ArrayList();  // for Students
list1.add(“test”);
list2.add(new Integer(17));
list2.add(new Integer(42));
...
list3.add(new Student());
list1.add(new Student());
list2.add(new Integer(4));
list1.add(“another string”);
```

Downside of downcasting

- Consider a method in which you use several collections to store data of several types:

```
ArrayList list1, list2, list3;
list1 = new ArrayList();  // for Strings
list2 = new ArrayList();  // for Integers
list3 = new ArrayList();  // for Students
list1.add("test");
list2.add(new Integer(17));
list2.add(new Integer(42));
...
list3.add(new Student());
list1.add(new Student());  // Wrong list!
list2.add(new Integer(4));
list1.add("another string");
```

Downside of downcasting

- If we later retrieve an `Object` from `list1` and assume (incorrectly) that it contains only `String`s, our program will crash:

```
final Iterator iterator = list1.iterator();
while (iterator.hasNext()) {
  final String s = (String) iterator.next();
  ...
}
```

Given the code on previous slide, this will trigger a `ClassCastException`.

- It is still nice that the JVM catches our mistake at *run-time*, but it would be even *nicer* for the Java compiler to catch our mistake at *compile-time*.

Downside of downcasting

- Unfortunately, with collections of `Objects`, this is not really possible.
- The compiler has no way of "knowing" that `list1` was intended "only for `Strings`".
  - `ArrayList.add(o)` is happy to accept any `Object o`.

More plausible example

- A more plausible example of the problem above might occur if you are implementing a method that takes a collection as a *parameter*:

```
// Specified list should contain only Strings.
// Returns ArrayList of appended strings.
List12 appendStrings (List12 strList) {
  List12 appendedStrList = new ArrayList();

  final Iterator iterator = strList.iterator();
  while (iterator.hasNext()) {
    appendedStrList.add(
      "appendage" + (String) list.next()
    );
  }
  return appendedStrList;
}
```

More plausible example

- A more plausible example of the problem above might occur if you are implementing a method that takes a collection as a *parameter*:

```
// Specified list should contain only Strings.
// Returns ArrayList of appended strings.
List12 appendStrings (List12 strList) {
  List12 appendedStrList = new ArrayList();

  final Iterator iterator = strList.iterator();
  while (iterator.hasNext()) {
    appendedStrList.add(
      "appendage" + (String) list.next()
    );
  }
  return appendedStrList;
}
```

If user passed in ArrayList that contained any non-`String` object, then we'll get a `ClassCastException`.

Naive fix

- How can we fix the problems of tedium, ugly code, and potential `ClassCastExceptions`?
- One naive strategy is to define a different `ArrayList` for every class we want to store in it, e.g.:

With specific types, we no longer have to downcast the result, and we're guaranteed that `get(index)` returns a `String`.

```
class ArrayListOfStrings {
  public void add (String s) { ... }
  public String get (int index) { ... }
}
class ArrayListOfIntegers {
  public void add (Integer i) { ... }
  public Integer get (int index) { ... }
}
class ArrayListOfShapes {
  public void add (Shape s) { ... }
  public Shape get (int index) { ... }
}
```

Naive fix

- However, this "naive fix" is very tedious -- we have to create another version of the **`ArrayList`** for every class we want to support.
- "Copying+pasting code" would save some time, but this is never a good idea.
  - Inevitably, one of the **`ArrayListOfX`** classes will change, and you'll forget to change the other ones correspondingly.
- Let's take another look at those "related classes"...

Better fix: factor out the type

```
class ArrayListOfStrings {
  public void add (String s) { ... }
  public String get (int index) { ... }
}
class ArrayListOfIntegers {
  public void add (Integer i) { ... }
  public Integer get (int index) { ... }
}
class ArrayListOfShapes {
  public void add (Shape s) { ... }
  public Shape get (int index) { ... }
}
```

- The *only* place these class definitions differ is in the *type* of the objects they hold.
- It seems like there should be a way to “factor out” the type...

Java generics.

Java generics

- Since Java 1.5, Java has offered the ability to parameterize a class by a type.
  - For example, when writing a "collection" class such as **`ArrayList`**, we can give it a type parameter **`T`**.
    - As with data parameters, the type parameter is up to the "user" programmer.
    - Type parameters are typically given one-letter names:
      - **`K`** for "key", **`V`** for "value", **`E`** for "element, etc.

Generics for "`ArrayListOfX`"

- Consider our problem of writing multiple **`ArrayListOfX`** classes to store data of different types:
  - With Java generics, we can write just one version of the class and parameterize it by type **`T`**, the type of data the **`ArrayList`** should contain.

Generics for "ArrayListOfX"

```
class ArrayList<T> implements List<T> {
  T[] _underlyingStorage;
  int _numElements;

  void add (T element) {
    _underlyingStorage[_numElements] = element;
    _numElements++;

  T get (int index) {
    return _underlyingStorage[index];
  }
}
```

Interfaces too can be parameterized by a type.

The type parameter **T** is specified in angled brackets just after the classname. Thereafter, it can be used inside the class anywhere a type is expected. (Almost -- more later.)

Generics for "ListOfX"

- Similarly to classes, interfaces too can be parameterized by a type:

```
interface List<T> {
  void add (T element);
  T get (int index);
  void remove (int index);
}
```

Generics for "**ArrayListOfX**"

- In short: (almost) everywhere in our previous versions of **List** and **ArrayList**, we replace the type **Object** with the type parameter **T**.

- To instantiate the "generic" **ArrayList<T>** in code:

  When we instantiate the generic collection, we must specify the *value* of the type parameter.

```
ArrayList<Student> list = new ArrayList<Student>();
```

- Instantiating the **ArrayList** with type parameter **T=Student** can be *conceptualized* as doing a "search-and-replace" to change **<T>** to **<Student>**:

```
class ArrayList<T> ... {
  void add (T element) { ...
  }
  T get (int index) { ...
  }
}
```

→

```
class ArrayList ... {
  void add (Student element) { ...
  }
  Student get (int index) { ...
  }
}
```

Generics for "`ArrayListOfX`"

- Now, our list can *only* be populated with **`Student`** data (or any *subclass* of **`Student`**):

```
list.add(new Student());  // -- ok by definition
list.add(new UCSDStudent());  // -- ok if it's a subclass
```

Upcast

- What happens if we try to break this rule and add a non-**`Student`** object to list?

```
list.add("error");  // not ok -- compiler catches this!
```

Generics for "ListOfX"

- Now, our list can *only* be populated with **`Student`** data (or any *subclass* of **`Student`**):

```
list.add(new Student());  // -- ok by definition
list.add(new UCSDStudent());  // -- ok if it's a subclass
```

- What happens if we try to break this rule and add a non-**`Student`** object to list?

```
list.add("error");  // not ok -- compiler catches this!
```

- With Java generics, the compiler will catch this error -- it knows that "error" is a **`String`**, and that list is of type **`ArrayList<Student>`**.
    - Since **`ArrayList<Student>`**'s **`add(element)`** method expects a **`Student`**, there is a type mismatch -- *compile-time* error.

Benefits of generics

- It is preferable for the compiler to catch this mistake rather than the JVM:
  - We fix the bug *before* the program crashes.
  - The compiler *rules out the possibility* that we mismatch container type and element type.
- With generics, we also no longer have to *downcast* the return value of **`get(index)`**:

```
final ArrayList<String> list = new ArrayList<String>();
list.add("hello");
final String s = list.get(0);  // No downcast necessary
```

  - This is because the result of **`get(index)`** is *guaranteed* to be of type **`String`** -- we don't have to additionally "promise" the compiler anything.

Making `List` generic

- The benefits of a generic *interface* are exactly analogous to the benefits of a generic *class*:
  - When you use a variable of the interface type, the compiler will check that the types are consistent:

```
final List<Integer> list = ... // some concrete impl
list.add(new Integer(5));  // ok
list.add("test");  // not ok -- compile-time error
```

Making List generic

- As described before, when writing a generic **`List`**, we include a type parameter at the start of the class definition.
  - The type parameter tells the generic **`List`** interface which type of element the list can accept.
- We can define a generic List interface as follows:

```
interface List<T> {
  int size ();
  void add (T element);
  T get (int index);
  void remove (int index);
}
```

Generics syntax

- Let's examine more carefully how the syntax works:

```
interface List<T> {
  int size ();
  void add (T element);
  T get (int index);
  void remove (int index);
}
```

Generics syntax

- Let's examine more carefully how the syntax works:

```
interface List<T> {
  int size ();
  void add (T element);
  T get (int index);
  void remove (int index);
}
```

When we write angled brackets just after the type name, we are *declaring* a *type parameter* and giving it the name `T`.

This is analogous, in a Java method signature, to declaring a data parameter and giving it the name `student`.

```
void method (Student student) {
  student.setAge(24);
  student.printAddress();
}
```

Generics syntax

- Let's examine more carefully how the syntax works:

```
interface List<T> {
  int size ();
  void add (T element);
  T get (int index);
  void remove (int index);
}
```

Following the declaration of type parameter T, whenever we write T, we are *using* the type parameter's *value*.

This is analogous, in a Java method signature, to *using* that parameter inside the method body.

```
void method (Student student) {
  student.setAge(24);
  student.printAddress();
}
```

Generics syntax

- Now, suppose we want the `List` interface to extend the `Iterable` interface. We could write:

```
interface List<T> extends Iterable<T> {
  int size ();
  void add (T element);
  T get (int index);
  void remove (int index);
}
```

Despite the angled brackets, we are actually "using" `T`, not declaring `T`. We are "passing `T` to the generic `Iterable` interface."

This is analogous, in a Java method, to passing the parameter to another method

```
void method (Student student) {
  student.setAge(24);
  student.printAddress();
  otherMethod(student);
}
```

Generics syntax

- Bear in mind that *type parameters* are passed to a generic class at *compile-time*, whereas *data parameters* are passed to a method at *run-time*.

Making **`ArrayList`** generic

- Now that we have a generic **`List`**, we can define a generic **`ArrayList`**.
  - As mentioned last lecture, this consists mostly of replacing "**`Object`**" with "**`T`**":

```
class ArrayList<T> implements List<T> {
  T[] _underlyingStorage;
  int _numElements;

  public void add (T element) { ... }
  public T get (int index) { ... }
  public Iterator<T> iterator () {
  }

  private class ArrayListIterator implements Iterator<T> {
      ...
      T next () { ... }
      void remove () { ... }
      boolean hasNext () { ... }
  }
}
```

Making ArrayList generic

- There is one important exception, however:
  - In the constructor **`ArrayList()`**, we *cannot* write:

    `_underlyingStorage = new T[128];`

  - The Java compiler will give an error: "**`generic array creation`**".
  - It would also be illegal to try to write:
    `final T element = new T();`
  - Why?
    - It has to do with how generics are implemented "under the hood".

Erasure

- Java generics are implemented based on the principle of *erasure*.
- In one sentence:
  - After the Java compiler checks that the generic types are ok, it *erases* the type parameters associated with generic classes/methods and replaces them with just "`Object`". *

* Not quite true -- it actually replaces them with the upper bound of the type parameter.

Erasure

- Let's now define erasure more leisurely. Consider:

```
final List<String> list = new ArrayList<String>();
```

- The `List` was instantiated with a type parameter set to `String`.

- This means that `list.add(o)` now expects `o` to be of `String` type. It will then verify that variables passed to `add(o)` have the correct type:

```
list.add("yup");  // ok
list.add(new Object());  // will not compile
```

Erasure

- Now, after verifying that all type parameters are compatible with the generic `List`, the compiler proceeds to compile your code.
  - The compile strips away ("erases") all of the type parameters.
  - The code

    ```
    final List<String> list = new ArrayList<String>();
    ```

    is essentially replaced by:

    ```
    final List list = new ArrayList();
    ```

  - We're right back where we started -- an `List` of `Objects`!

Erasure

- Actually, not quite -- we still get two big benefits:
  - The compiler already verified that in all calls to `add(o)`, `o` was compatible with the `list`'s type.
    - No possibility of adding non-`String`s to `ArrayList` that's supposed to contain only `String`s.
  - We didn't have to cast the result of `get(index)` to be `String`.

Erasure

- However, the erasure does have some suboptimal side effects:
  - We cannot instantiate an object of generic type T:
    ```
    final T t = new T();  // won't compile
    ```
    - Reason: After stripping away the type information T, *the JVM wouldn't know which constructor to call.*
  - We also cannot instantiate arrays of generic type:
    ```
    final T[] array = new T[];  // won't compile
    ```

Arrays of generic type

```
final T[] array = new T[];  // won't compile
```

- As a work-around, we have to instantiate an array of a *particular* (non-generic) type. An array of **`Objects`** will actually be sufficient for **`ArrayList`**:

```
final T[] array = (T[]) new Object[128];
```

  - The ugly downcast is back.
  - However, we only have to do this once in all of **`ArrayList`**.
  - Since this one line of code is an implementation of **`ArrayList`**, the user need never be bothered by it.

Erasure

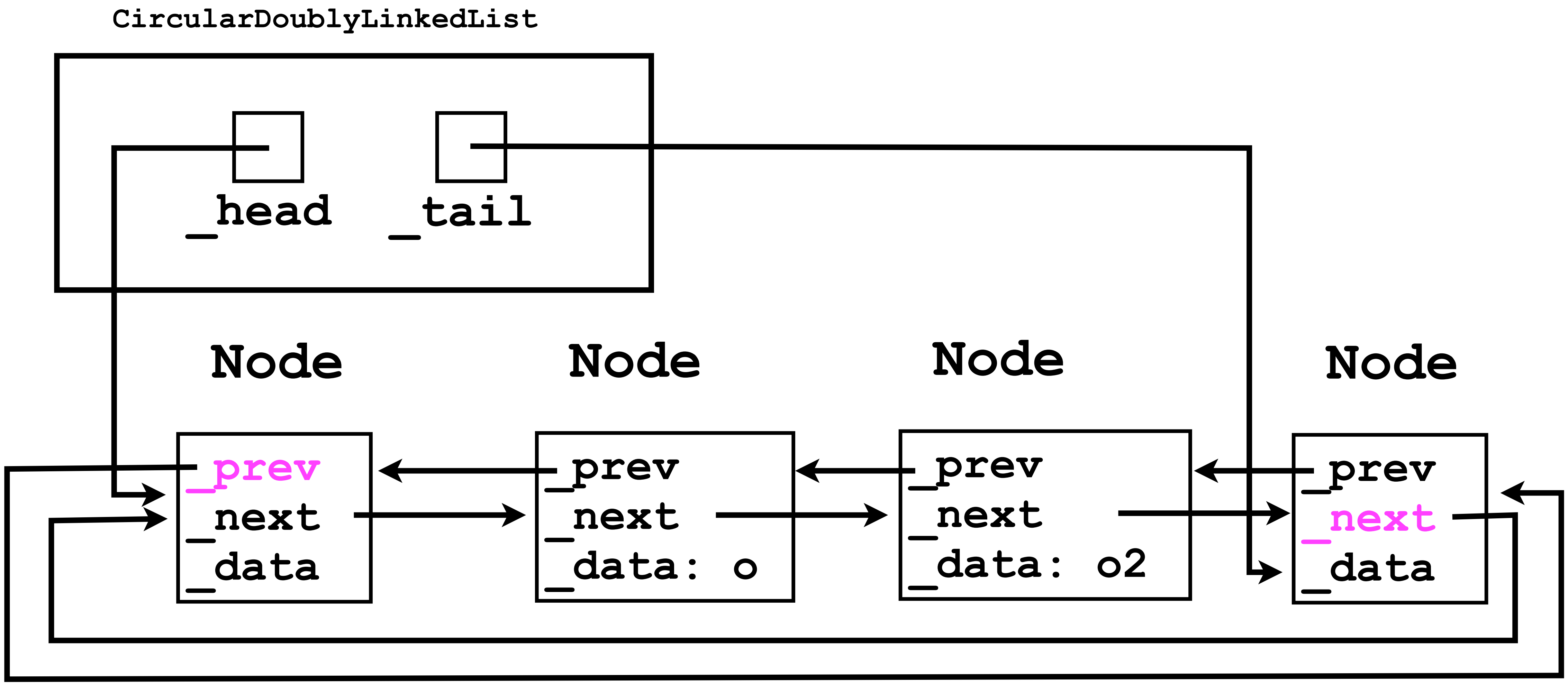
- FYI: C++ offers "templates" (analogous to generics).
  - Templates are not implemented using erasure.
    - Instead, the compiler essentially compiles a separate version of your generic class *for every type parameter you use*.
  - In C++, it is legal to write **`new T()`**;

Circular linked lists.

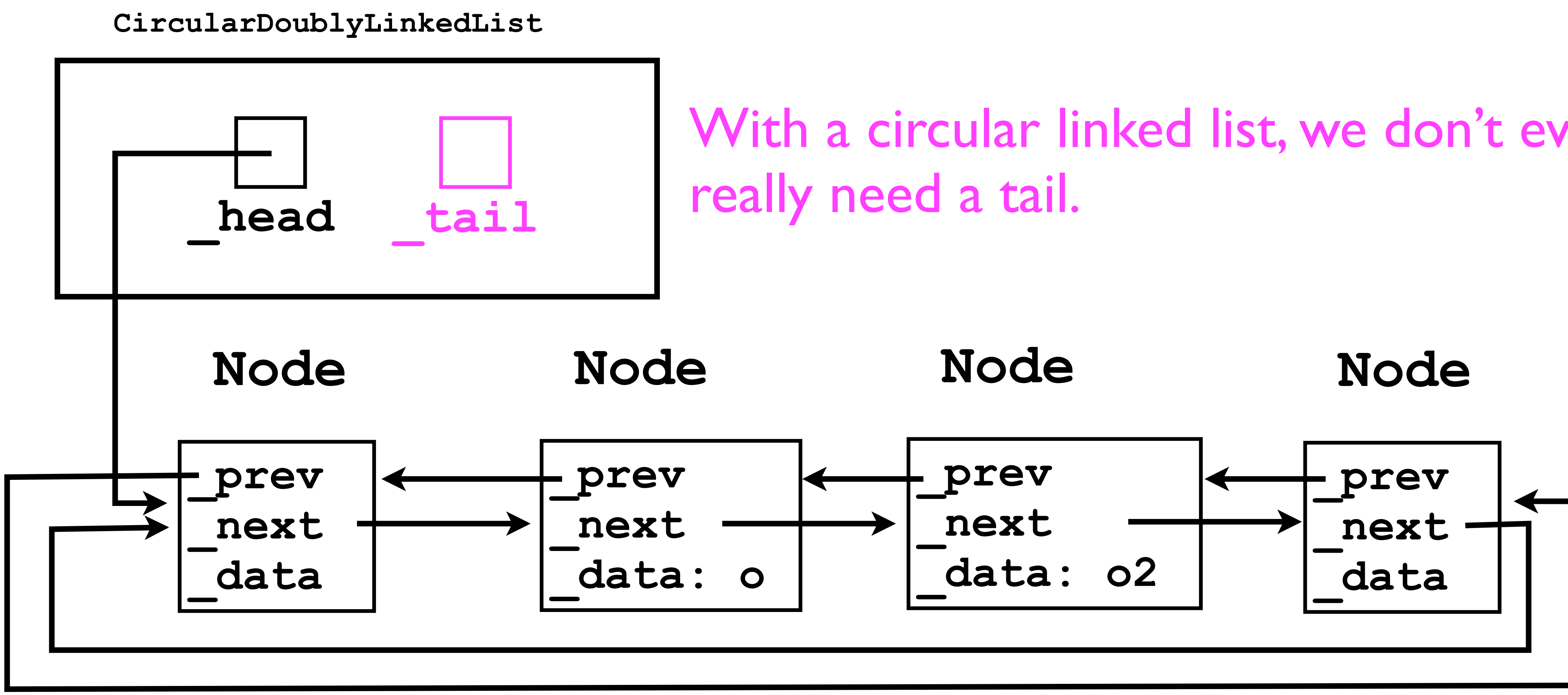
Circular linked lists

- Before moving on to other data structures, we will discuss one more variant of the basic "linked list" concept.
- A *circular linked list* is a list where the tail's "next" pointer points back to the *head*.
    - If the linked list is doubly-linked, then the head's "previous" pointer also points back to the *tail*.

Circular linked lists



Circular linked lists

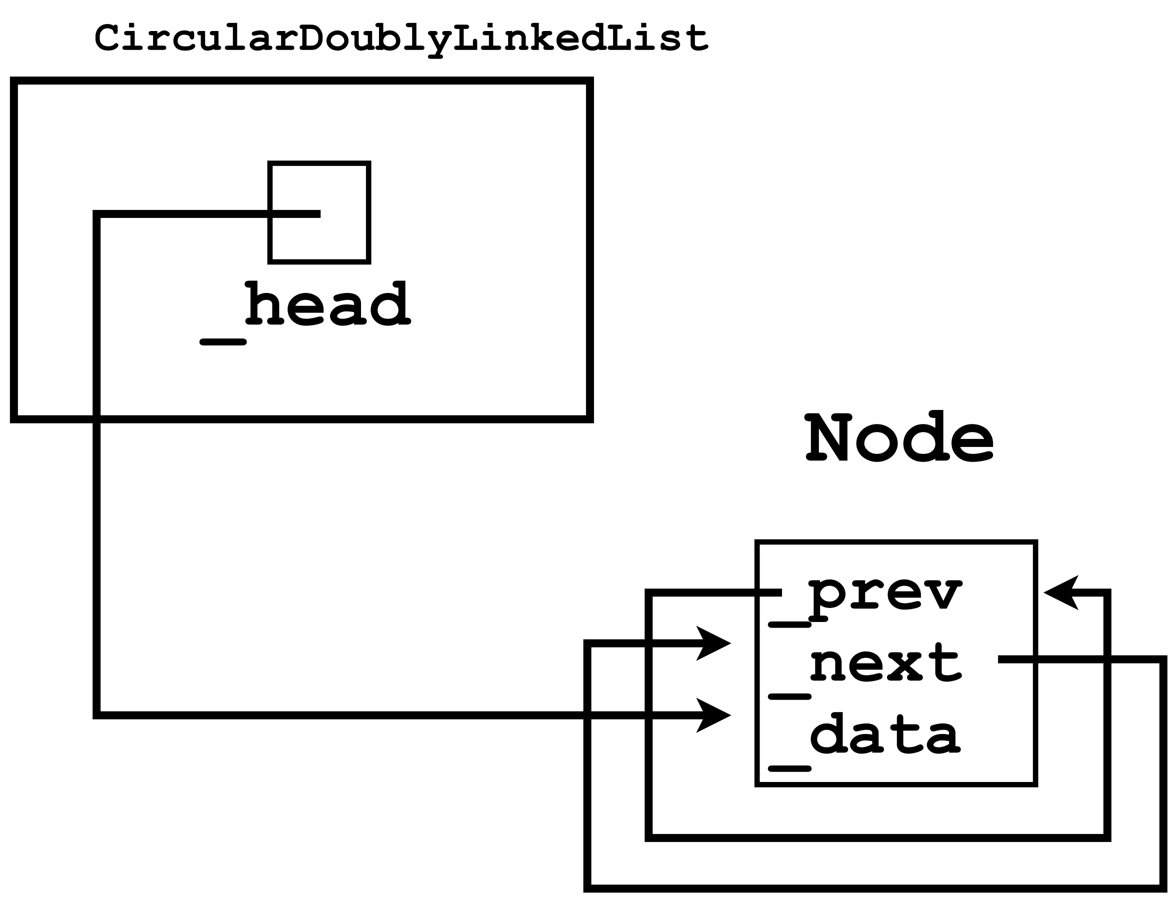


With a circular linked list, we don't even really need a tail.

Instead, all we really care about is whether we add to the front of the list (to the "right" of _head), or to the back of the list (to the "left" of _head).

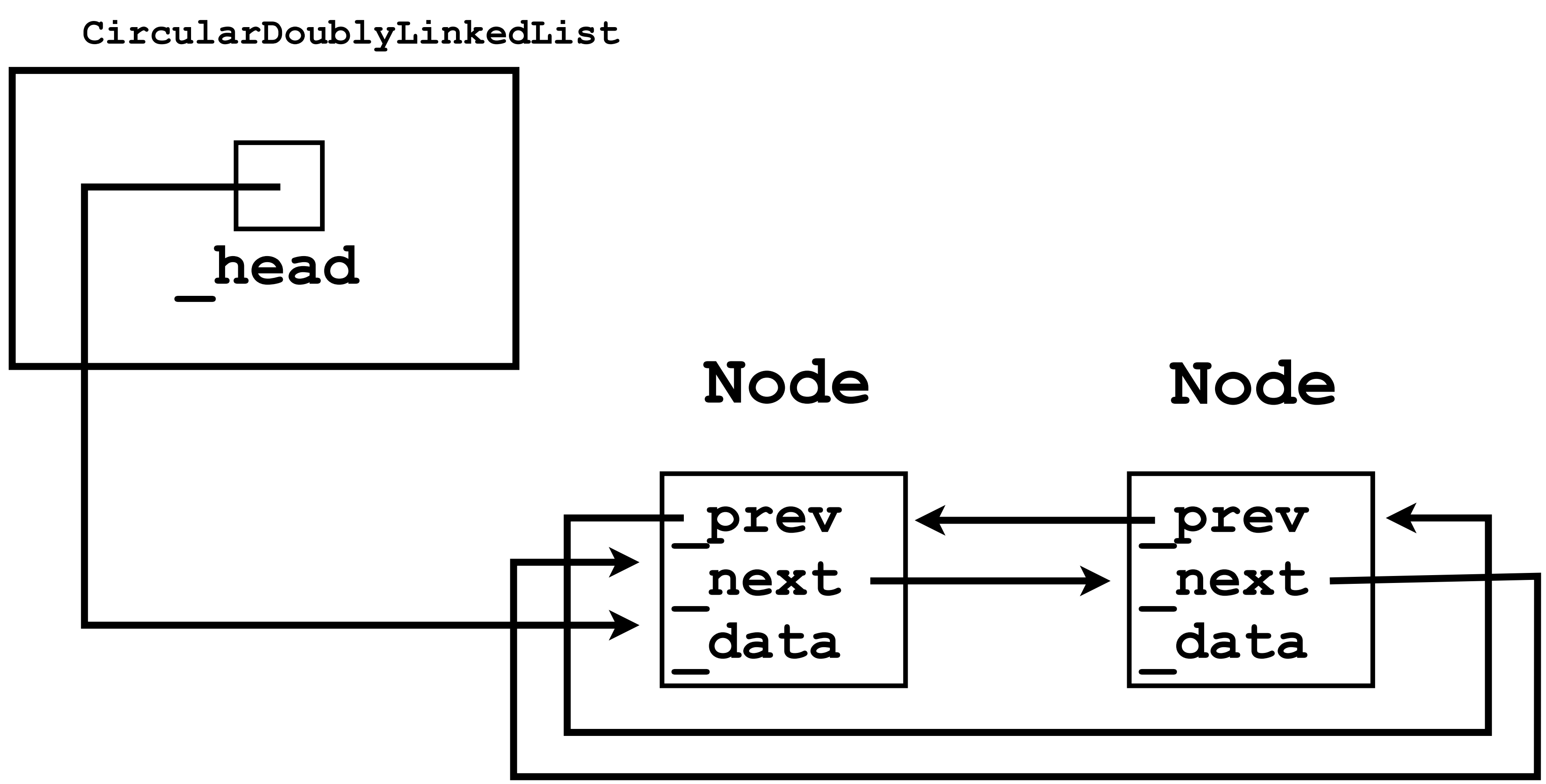
Circular linked lists

- The utility of circular linked lists is perhaps most clearly illustrated when there are no dummy nodes.
- Empty list: `_head = null`.
- List of size 1:



Circular linked lists

- List of size 2:



Iterating through a circular linked list

- As long as a circular linked list is non-empty, an **`Iterator`** can iterate *forever*.
  - Just keep following the current **`Node`**'s **`_next`** pointer.

```
class CircularListIterator {
  Node _current;
  ...
  boolean hasNext () {
    return _listSize > 0;
  }
  Object next () {
    _current = _current._next;
    return _current._data;
  }
}
```

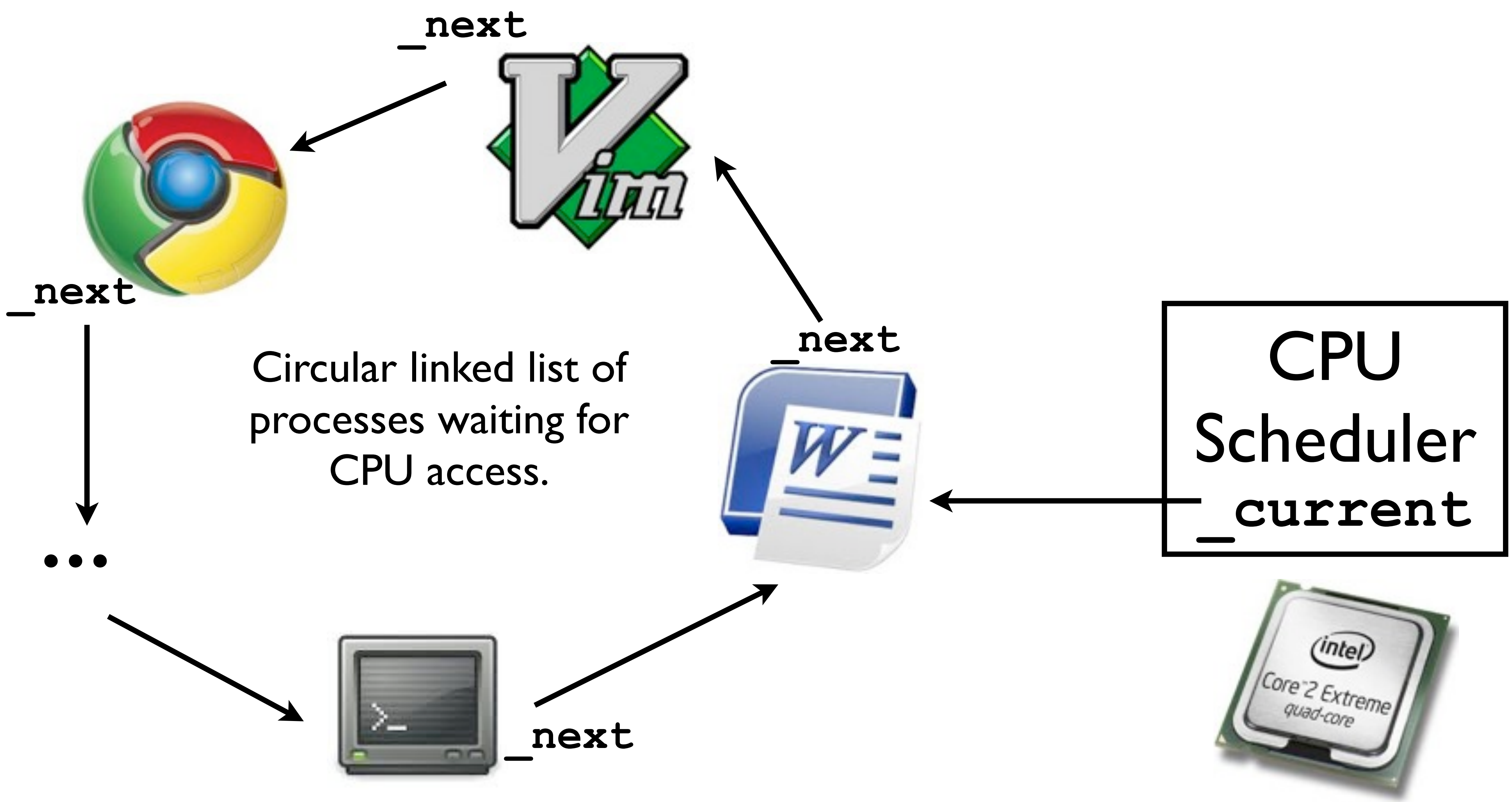
Circular linked lists

- Circular linked lists are most useful for storing a collection of objects in which "looping forever" is an intuitive and useful operation.
- Examples:
  - Looping around vertices of a polygon.

Circular linked lists

- CPU scheduling:
  - One CPU core can only execute one computer program at any given time.
  - On a single-core machine, to simulate "multitasking", each program is given a small "timeslice" (few milliseconds) to run on the CPU.
  - After the timeslice expires, the *next* program in the list of processes is selected, and so on.
  - After all programs in the list have received their timeslice, the CPU scheduler goes back to the first process.

Circular linked lists for CPU scheduling



Stacks and queues.

Stacks and queues.

- Let's now bring in two more fundamental data structures into the course.
- So far we have covered lists -- array-based lists and linked-lists.
  - These are both linear data structures -- each element in the container has at most one *successor* and one *predecessor*.
- Lists are most frequently used when we wish to store objects in a container, and *probably never remove them from it*.
  - E.g., if Amazon uses a list to store its huge collection of customers, it has no intention of "removing" a customer (except at program termination).

Stacks and queues

- Stacks and queues, on the other hand, are examples of *linear* data structures in which every object inserted into it will generally be removed:
  - The stack/queue is intended only as "temporary" storage.
- Both stacks and queues allow the user to add and remove elements.
- Where they differ is the *order* in which elements are removed *relative to when they were added*.

Stacks

- Stacks are *last-in-first-out* (LIFO) data structures.
- The classic analogy for a "stack" is a pile of dishes:
  - Suppose you've already added dishes A, B, and C to the "stack" of dishes.

If you try to remove a middle dish, you get that annoying clanging sound.

Stacks

- Stacks are *last-in-first-out* (LIFO) data structures.
- The classic analogy for a "stack" is a pile of dishes:
  - Suppose you've already added dishes A, B, and C to the "stack" of dishes.
  - Now you add one more, D.

If you try to remove a middle dish, you get that annoying clanging sound.

Stacks

- Stacks are *last-in-first-out* (LIFO) data structures.
- The classic analogy for a "stack" is a pile of dishes:
  - Suppose you've already added dishes A, B, and C to the "stack" of dishes.
  - Now you add one more, D.
  - Now you remove one dish -- *you get D back*.

If you try to remove a middle dish, you get that annoying clanging sound.

Stacks

- Stacks are *last-in-first-out* (LIFO) data structures.
- The classic analogy for a "stack" is a pile of dishes:
  - Suppose you've already added dishes A, B, and C to the "stack" of dishes.
  - Now you add one more, D.
  - Now you remove one dish -- *you get D back*.
  - If you remove another, you get C, and so on.
- With stacks, you can only add to/remove from the *top* of the stack.

If you try to remove a middle dish, you get that annoying clanging sound.

Usage example of stacks

```
Stack stack<String> = new Stack<String>();
stack.push("a");
stack.push("b");
stack.push("c");
stack.push("d");
...
String s;
s = stack.pop();  // returns "d"
```

push adds an object to the stack

pop both gets and removes the "last" object from the stack

Usage example of stacks

```
Stack stack<String> = new Stack<String>();
stack.push("a");
stack.push("b");
stack.push("c");
stack.push("d");
...
String s;
s = stack.pop();  // returns "d"
s = stack.pop();  // returns "c"
```

`push` adds an object to the stack

`pop` both gets and removes the "last" object from the stack

Usage example of stacks

```
Stack stack<String> = new Stack<String>();
stack.push("a");
stack.push("b");
stack.push("c");
stack.push("d");
...
String s;
s = stack.pop();  // returns "d"
s = stack.pop();  // returns "c"
s = stack.pop();  // returns "b"
```

push adds an object to the stack

pop both gets and removes the "last" object from the stack

Usage example of stacks

```
Stack stack<String> = new Stack<String>();
stack.push("a");
stack.push("b");
stack.push("c");
stack.push("d");
...
String s;
s = stack.pop();  // returns "d"
s = stack.pop();  // returns "c"
s = stack.pop();  // returns "b"
s = stack.pop();  // returns "a"
```

push adds an object to the stack

pop both gets and removes the "last" object from the stack

Stacks

- Stacks find many uses in computer science, e.g.:
  - Implementing *procedure calls*.
- Consider the following code:

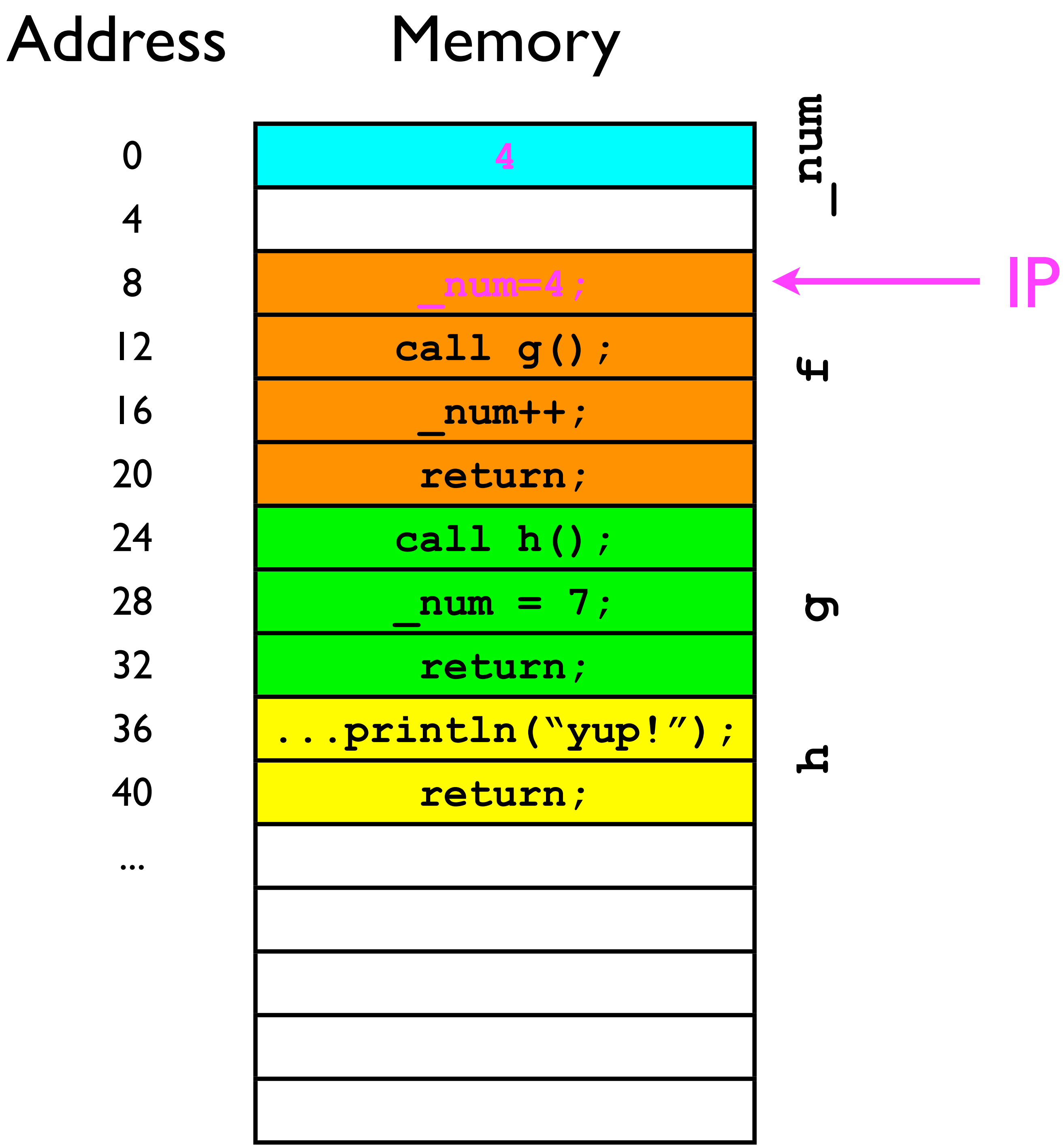
```
void f () {
  _num = 4;
  g();
  _num++;
}
void g () {
  h();
  _num = 7;
}
void h () {
  System.out.println("Yup!");
}
```

How does the CPU know to "jump" from `f` to `g`, `g` to `h`, then `h` back to `g`, and finally `g` back to `f`?

Von Neumann machine

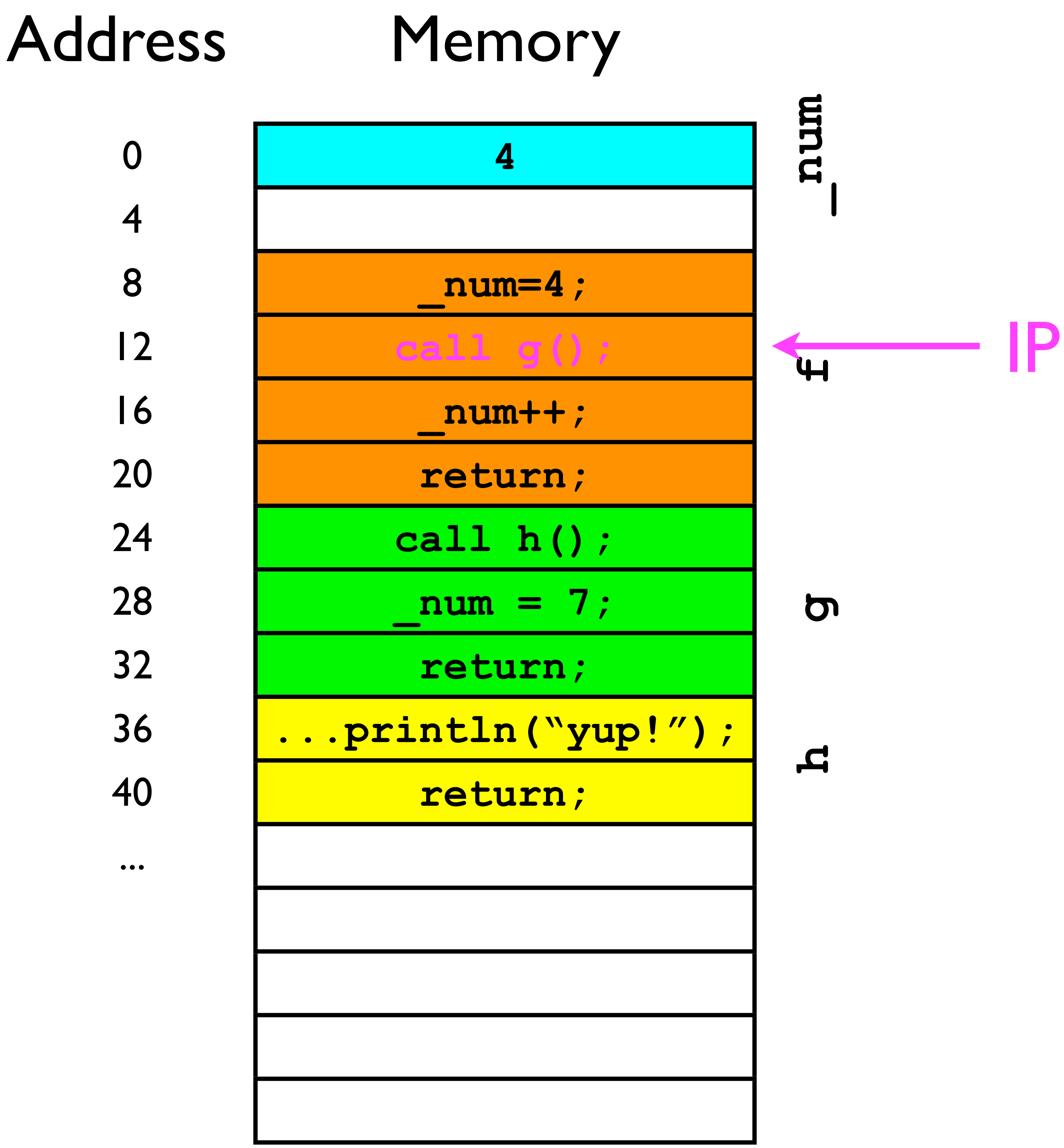
- On all modern machines, a program's *instructions* and its *data* are stored *together* somewhere in the computer's long sequence of bits (Von Neumann architecture).
  - Just by "glancing" at the contents of computer memory, one would have no idea whether a certain byte contains code or data -- it's all just bits.
- To keep track of which instruction in memory is currently being executed, the CPU maintains an Instruction Pointer (IP).

Code execution



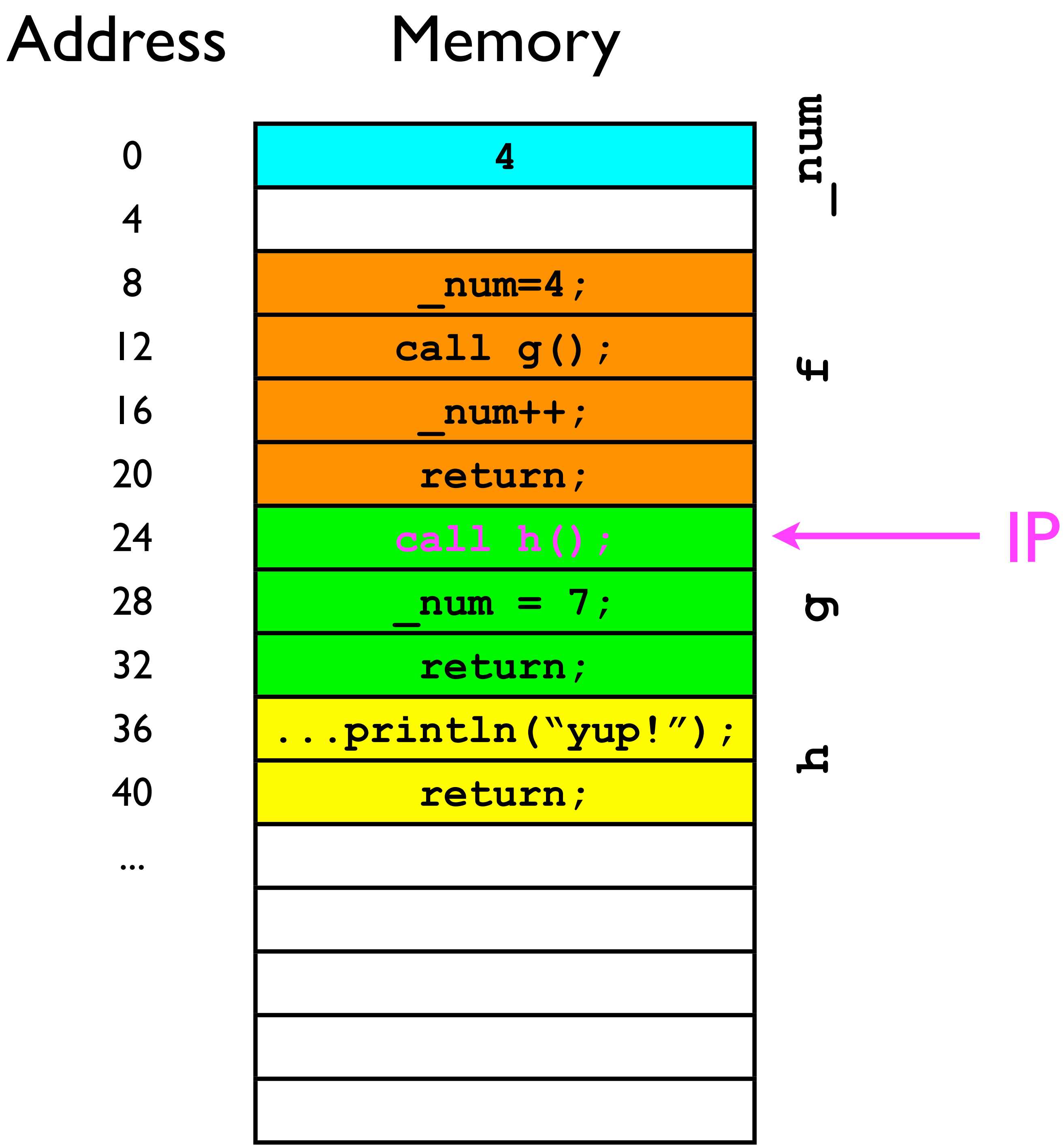
- Suppose the IP is 8:
  - Then the next instruction to execute is **_num=4**;
- The CPU then advances the IP to the next instruction (4 bytes later) to 12.

Code execution



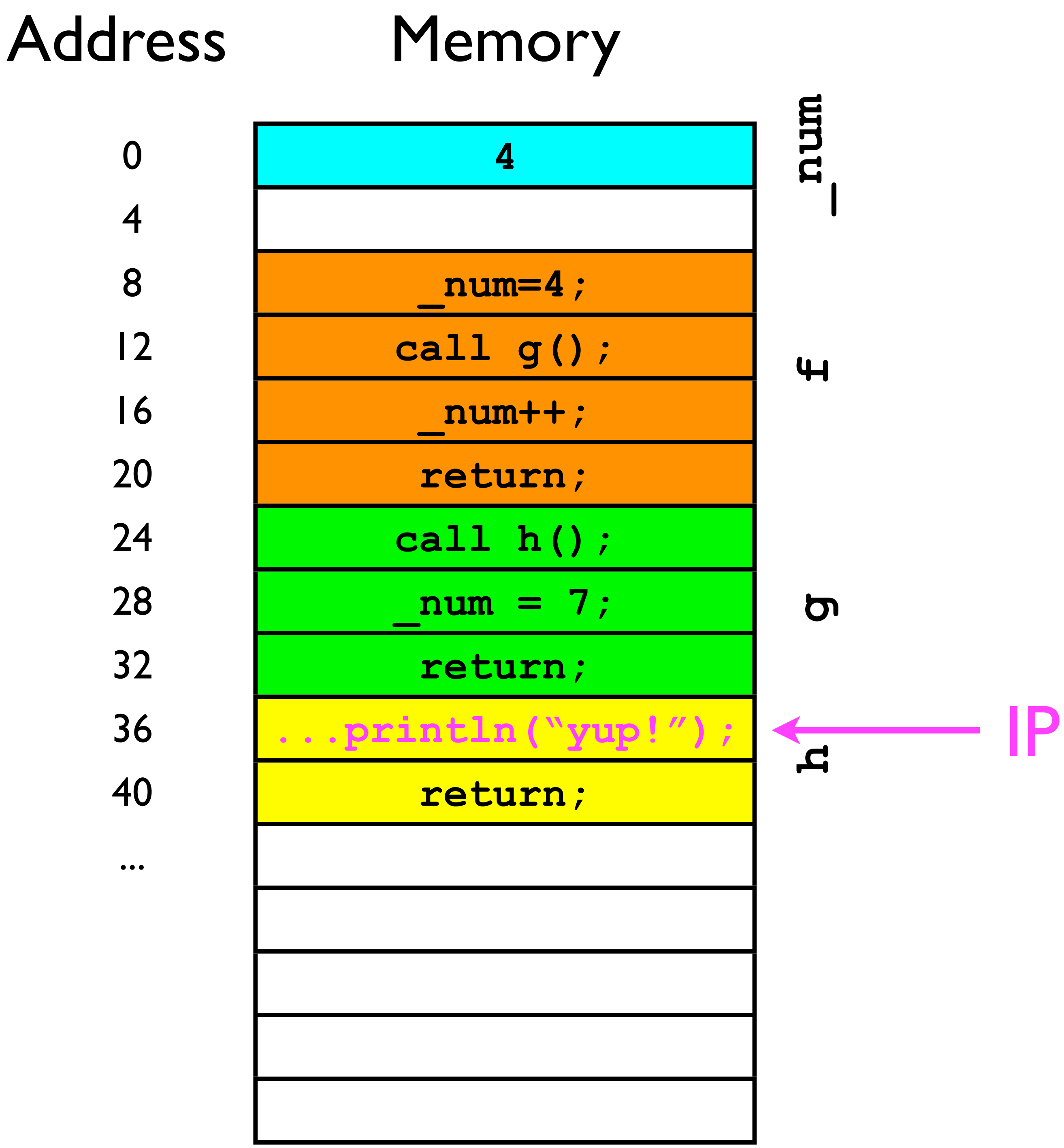
- The next instruction is `call g()`.
- The CPU must now "move" the IP to address 24 (start of g's code) so **g** can start.

Code execution



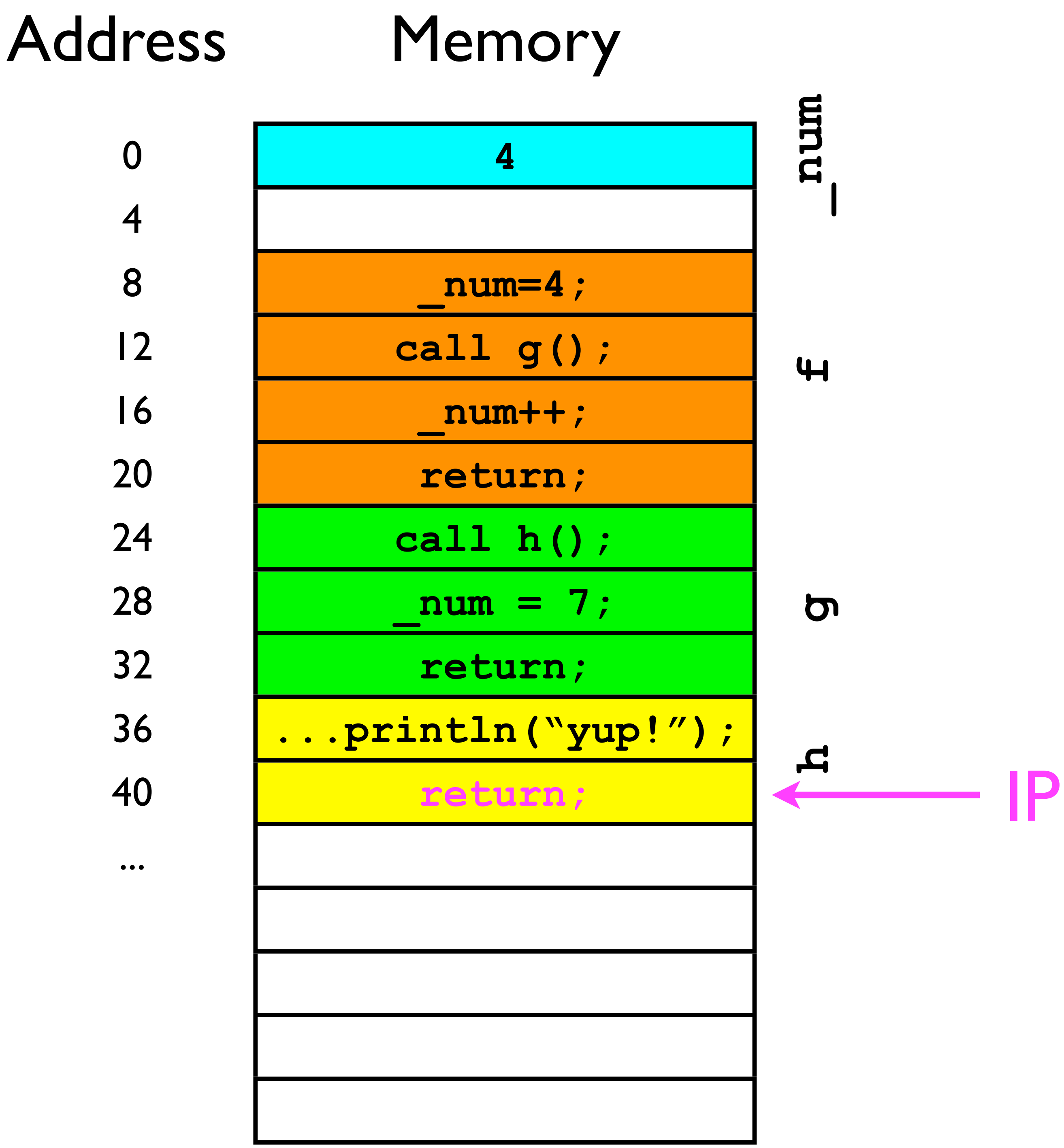
- **g** has now started.
- The first thing **g** does is call **h**.
- We have to move the IP again.

Code execution



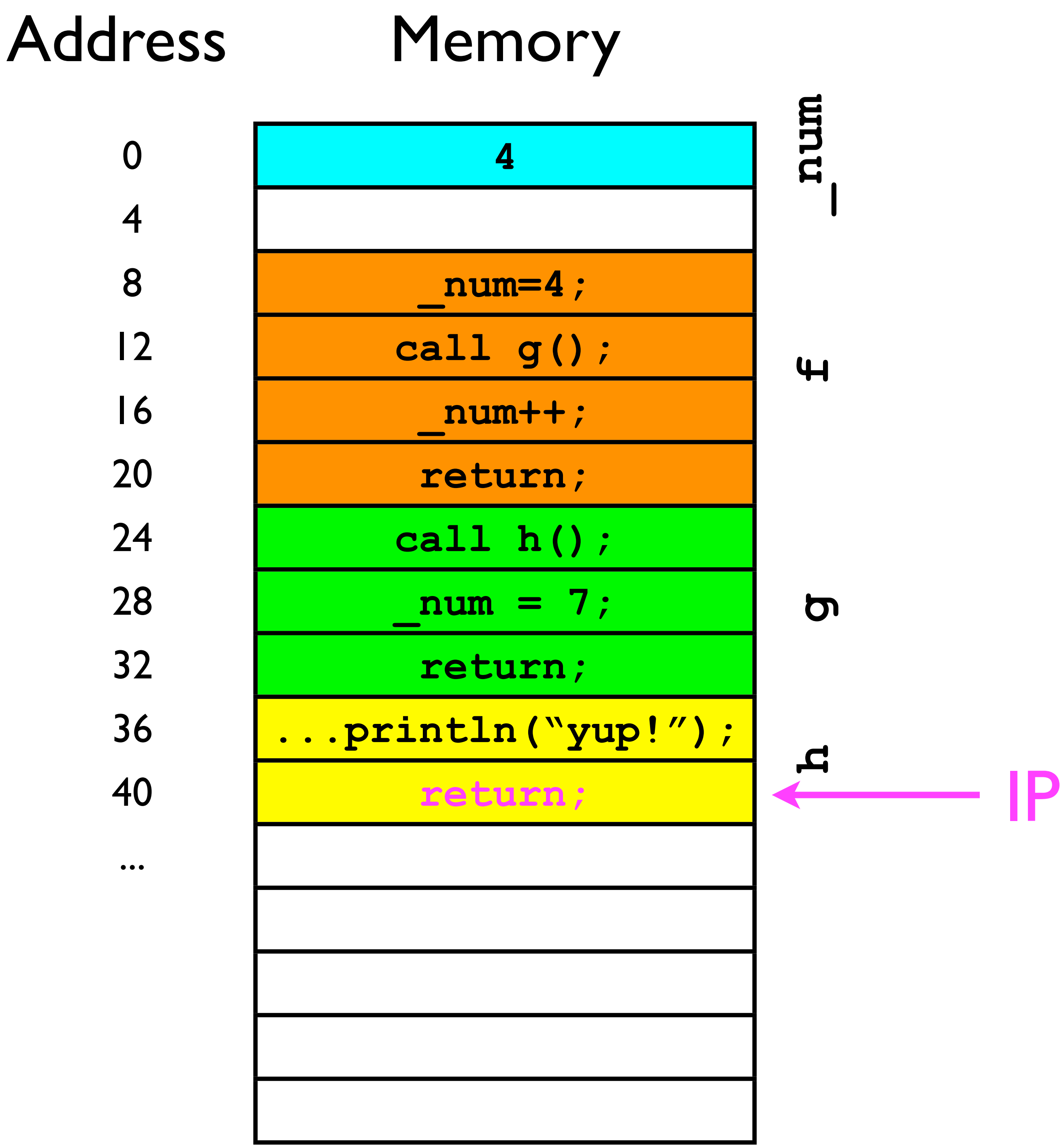
- **h** now prints out "yup!".

Code execution



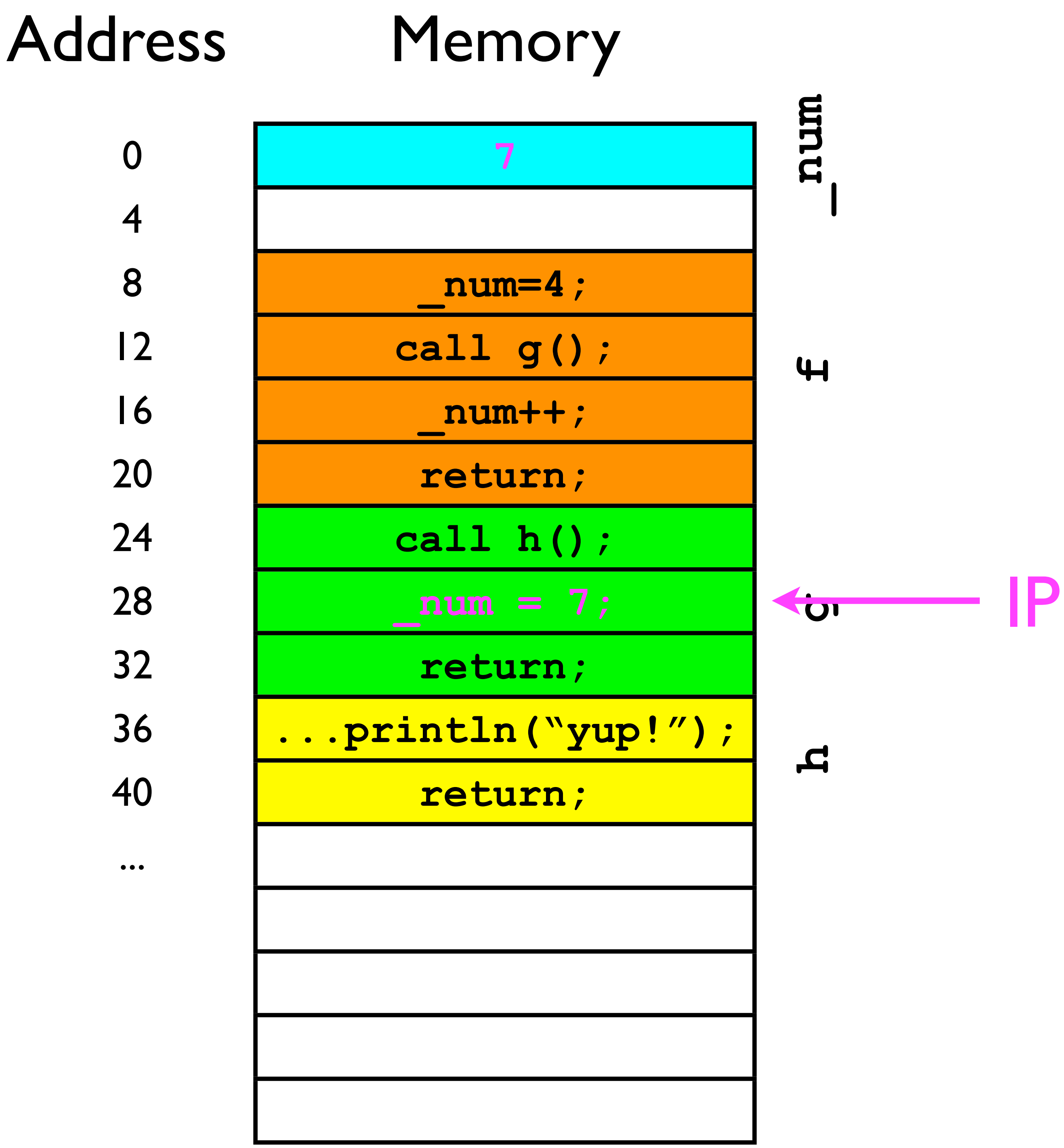
- The return instructions tells the CPU to move the IP back to where it was *before the current method was called.*
- But where is that?

Code execution



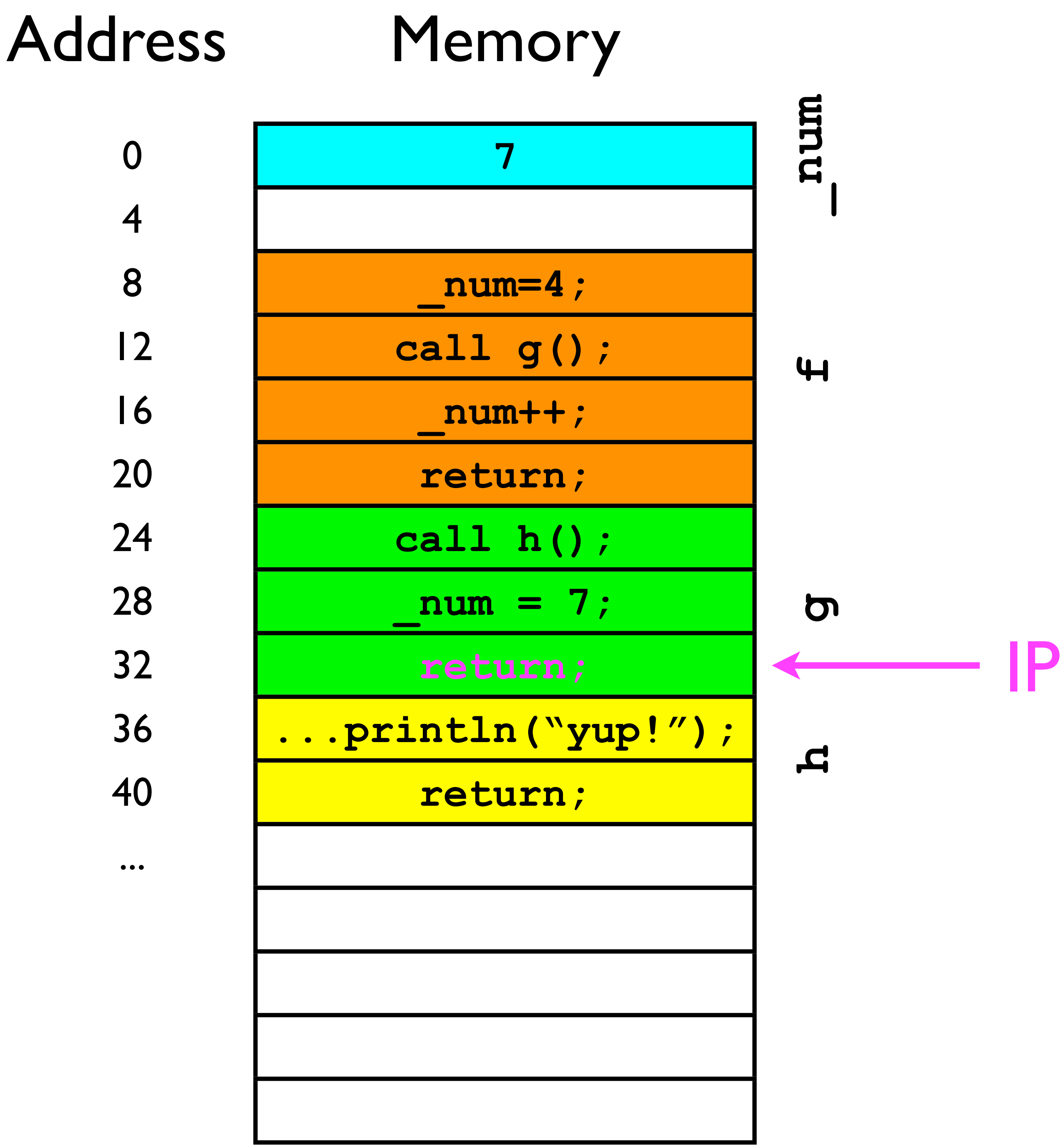
- The return call at address 40 *should* cause the CPU to jump to address 28 -- *the next instruction in* g.

Code execution



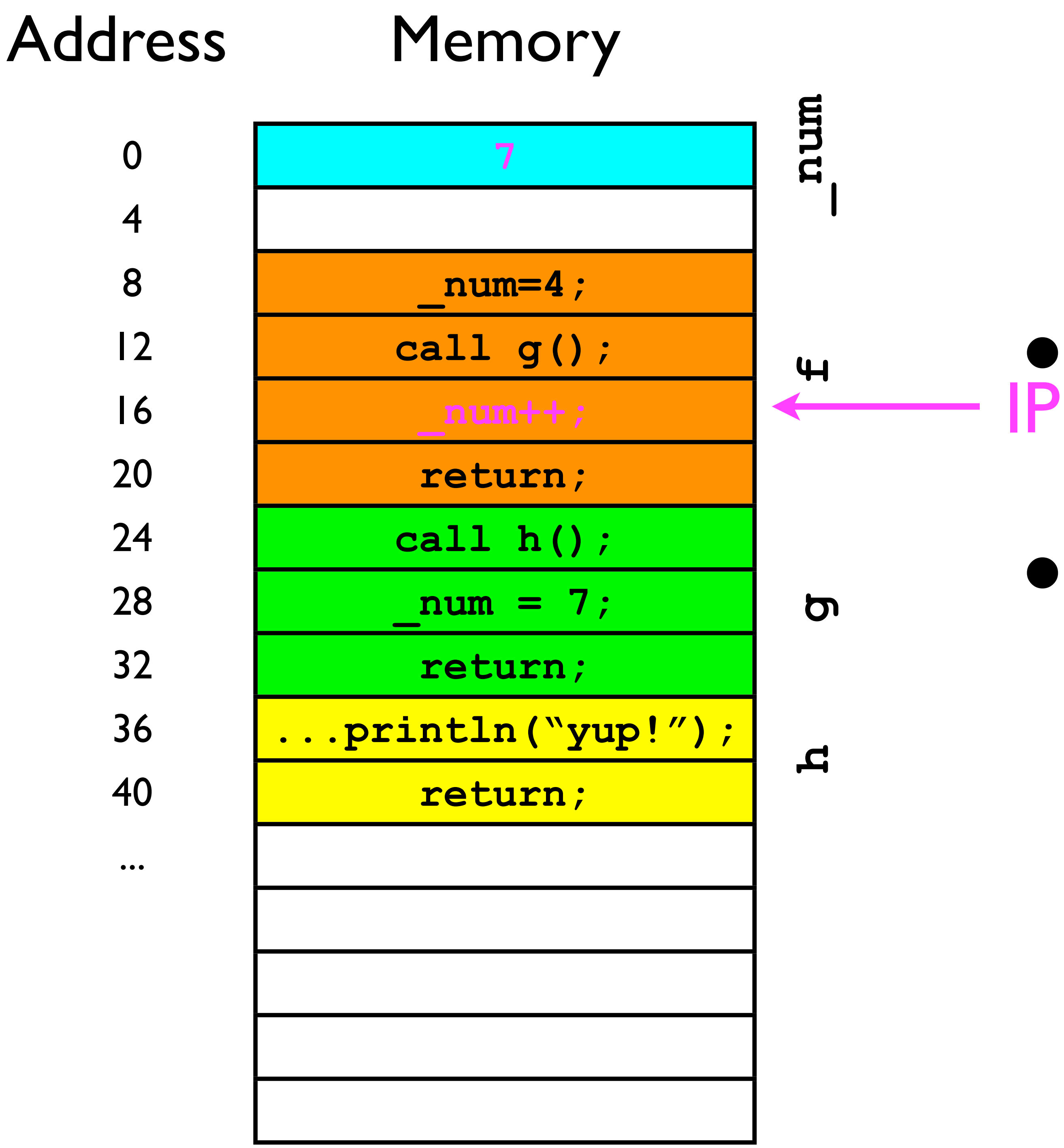
- We then execute `_num=7;`

Code execution



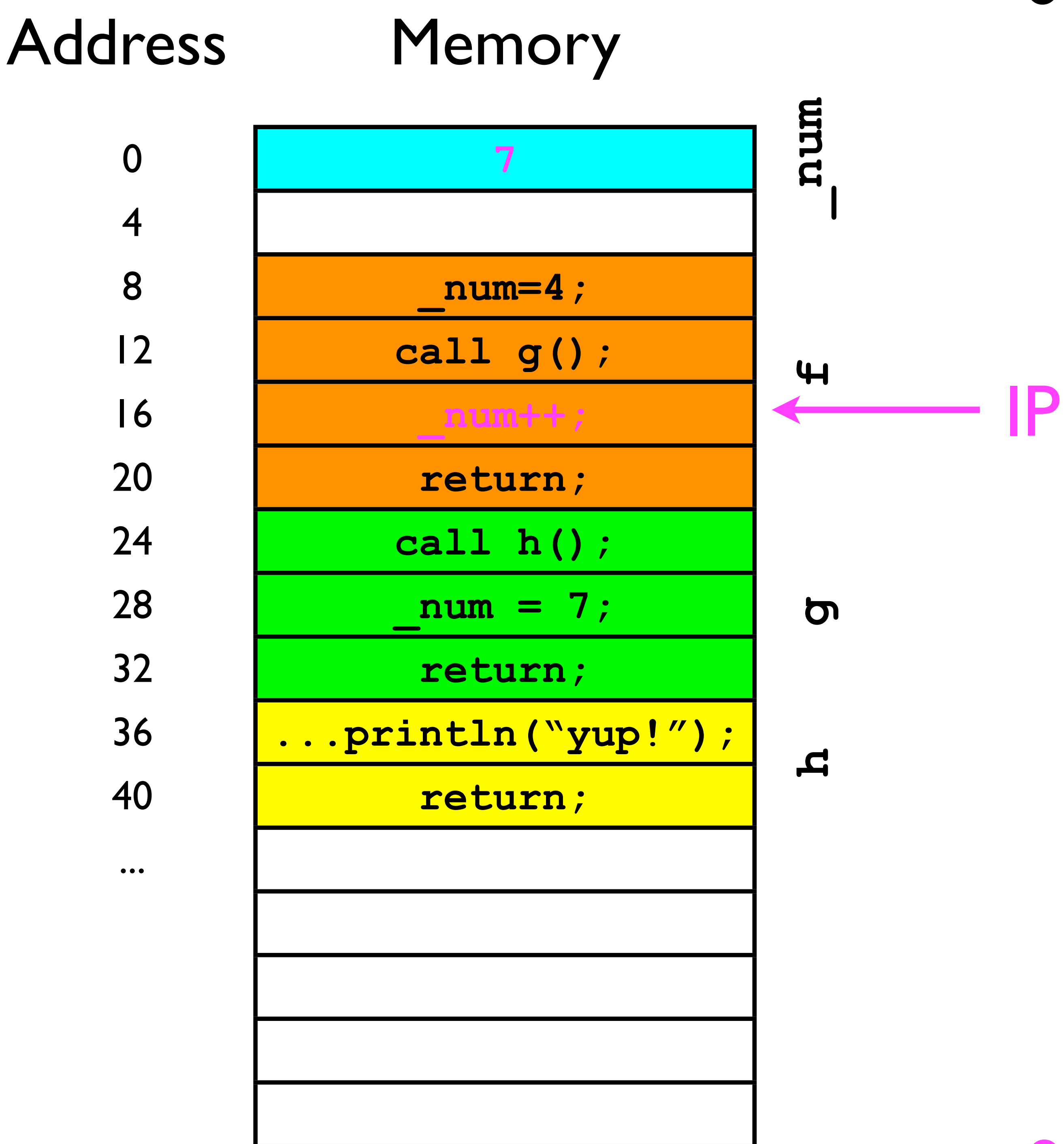
- And now we have to return to where the *caller* of g left off (address 16).

Code execution



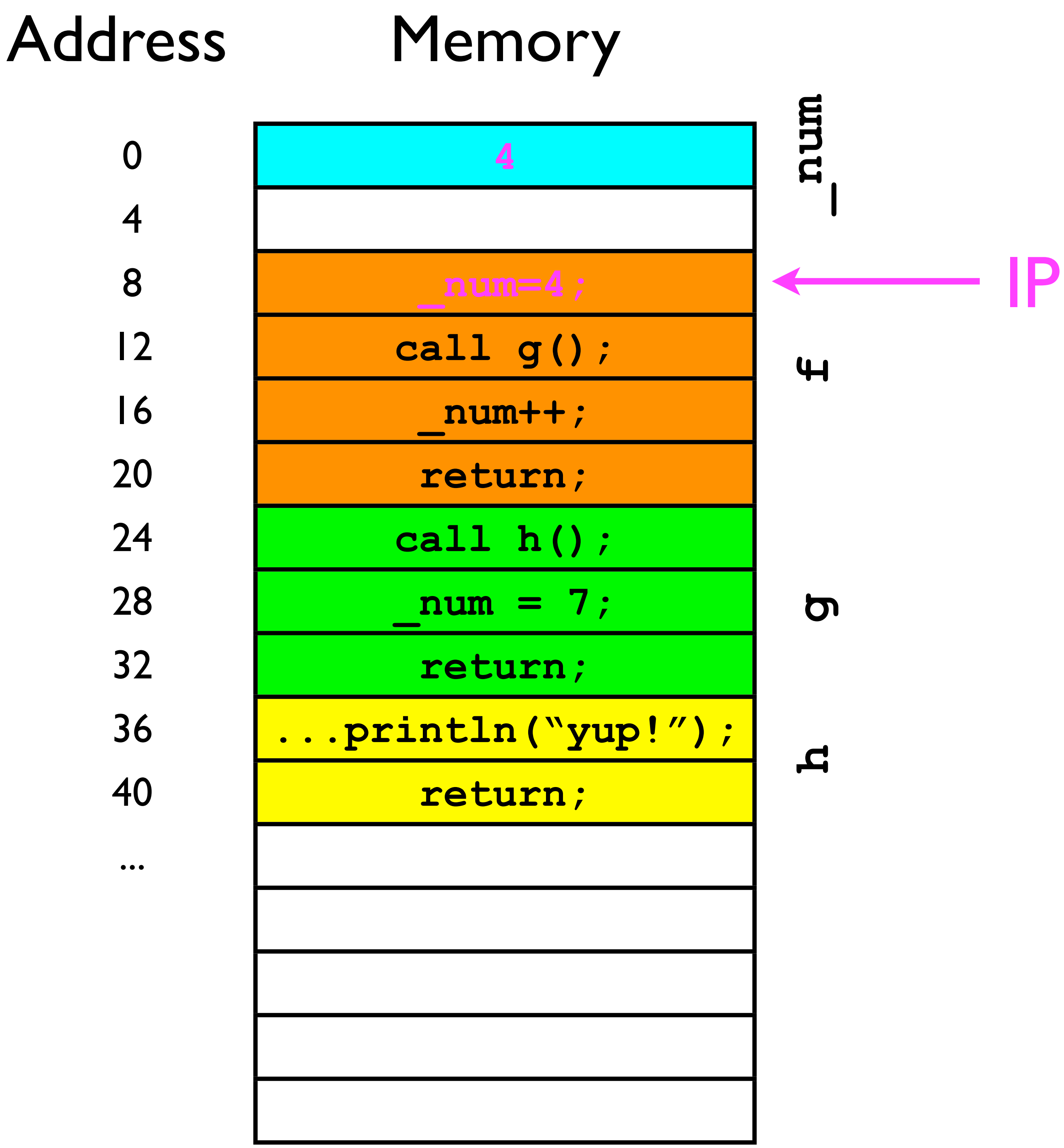
- How does the CPU know which address to "return" to?
- We need some kind of data structure to manage the "return addresses" for us.

Code execution



- What we need is a last-in-first-out data structure ("stack") to remember all the return addresses:
  - *Rule 1*: Before method x calls method y, method x first adds its "return address" to the stack.
  - *Rule 2*: When method y "returns" to its caller, it removes the top of the stack and sets the IP to that address.
- Let's see this work in practice...

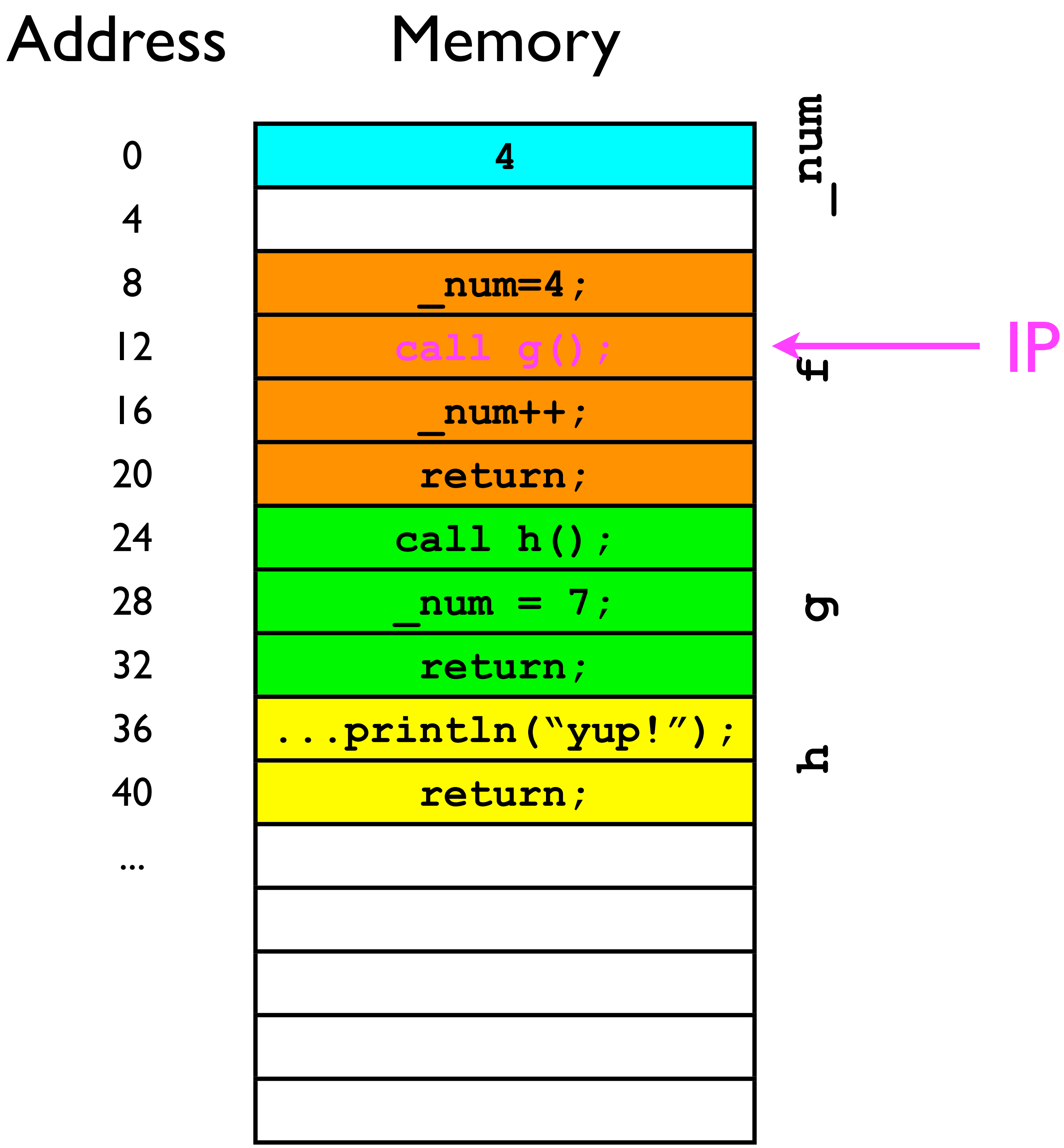
Code execution



- "Return address" stack:

(bottom of stack)

Code execution



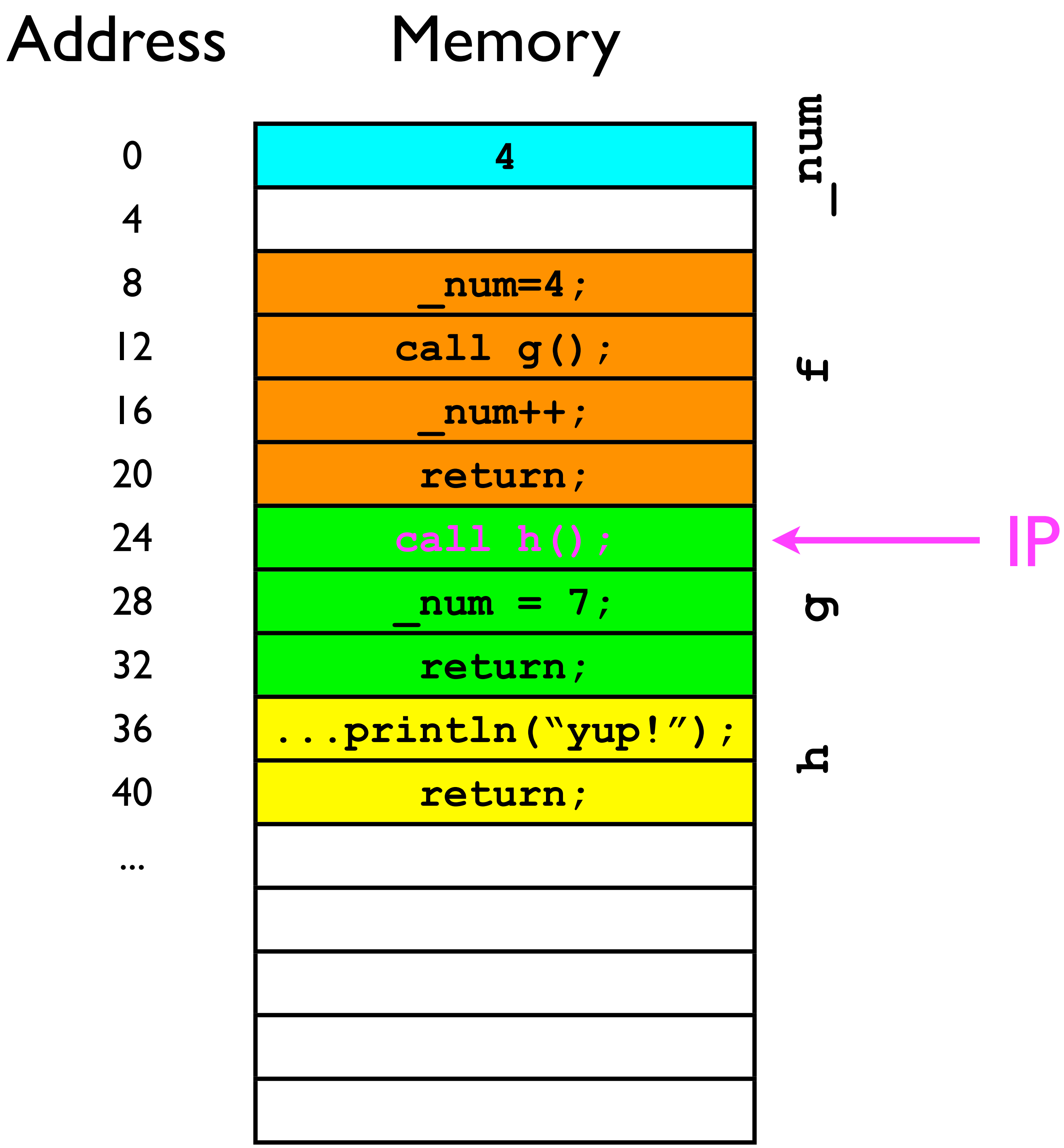
- "Return address" stack:

"push" 16 onto stack

16

(bottom of stack)

Code execution



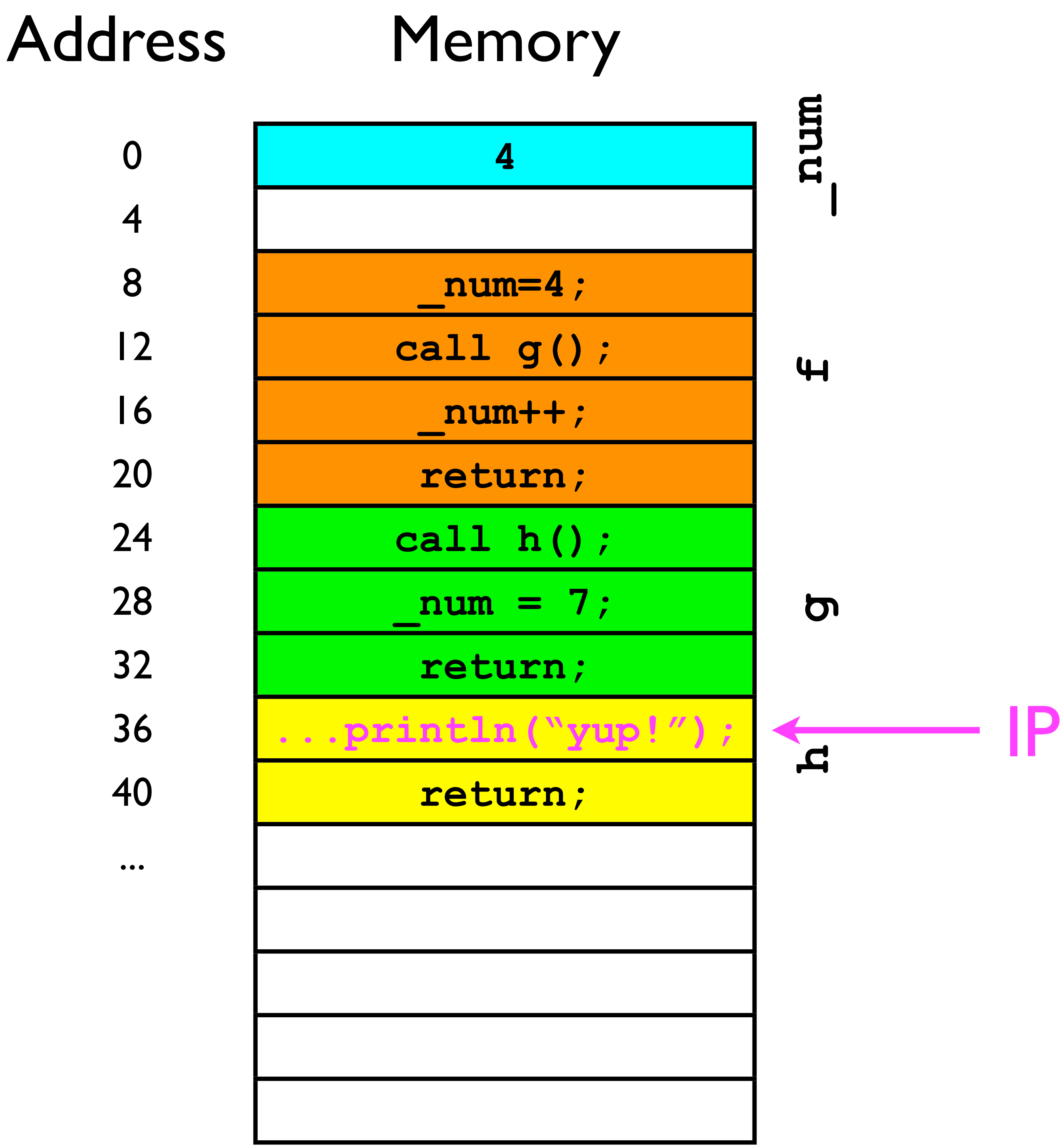
- "Return address" stack:

"push" 28 onto stack

28
16

(bottom of stack)

Code execution

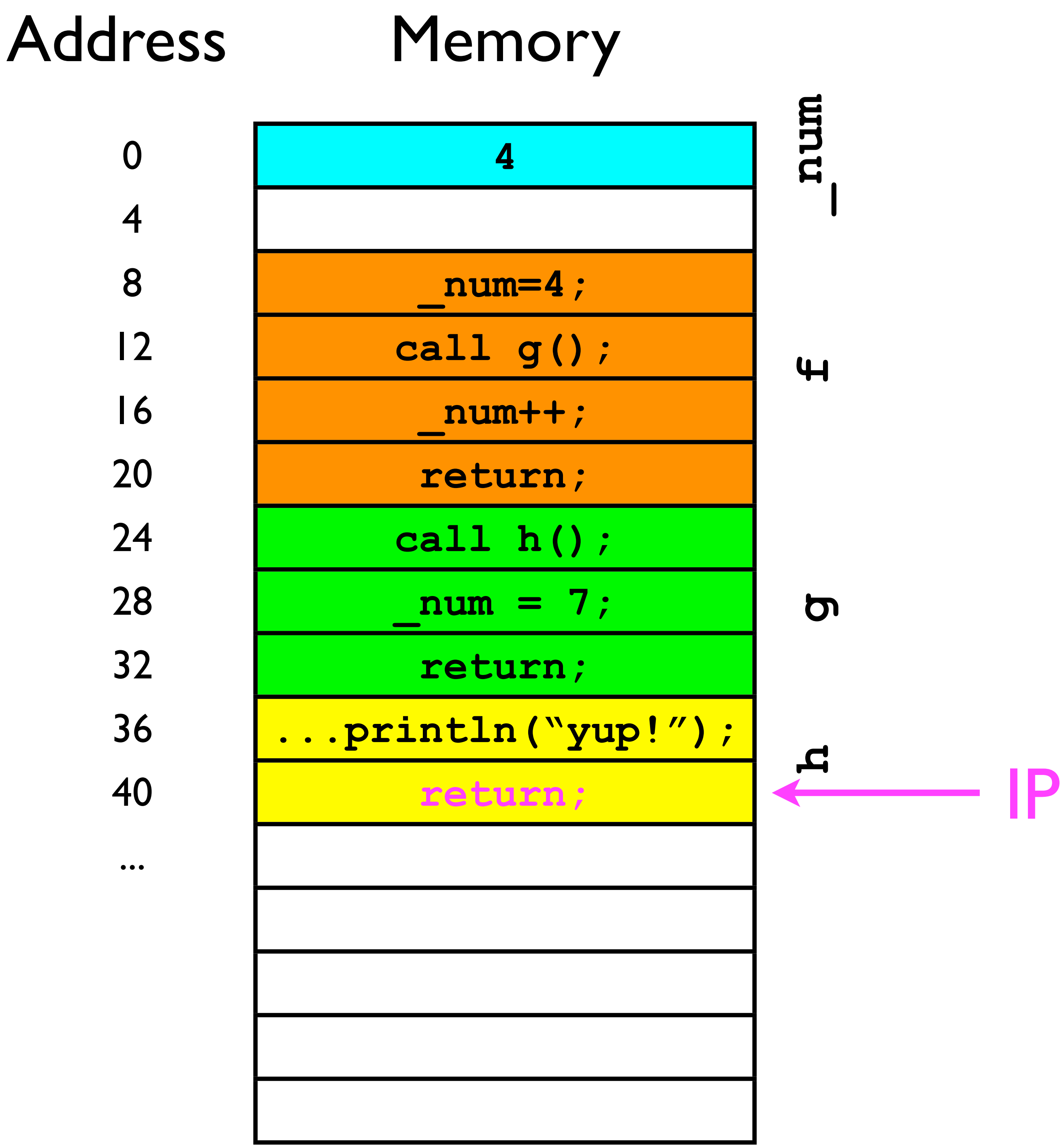


- "Return address" stack:

28
16

(bottom of stack)

Code execution

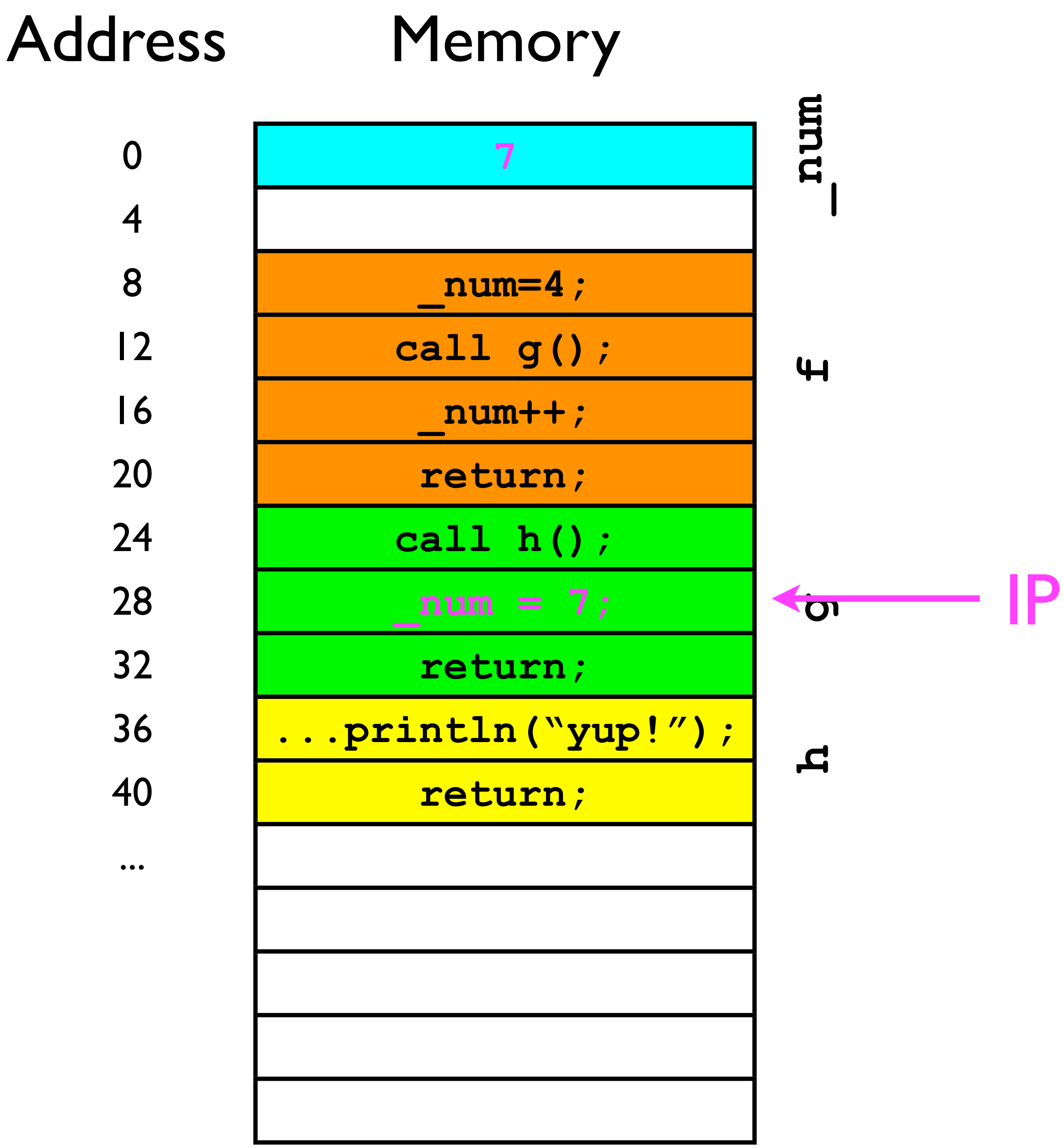


- "Return address" stack:

"pop" 28 off the stack...

28
16
(bottom of stack)

Code execution



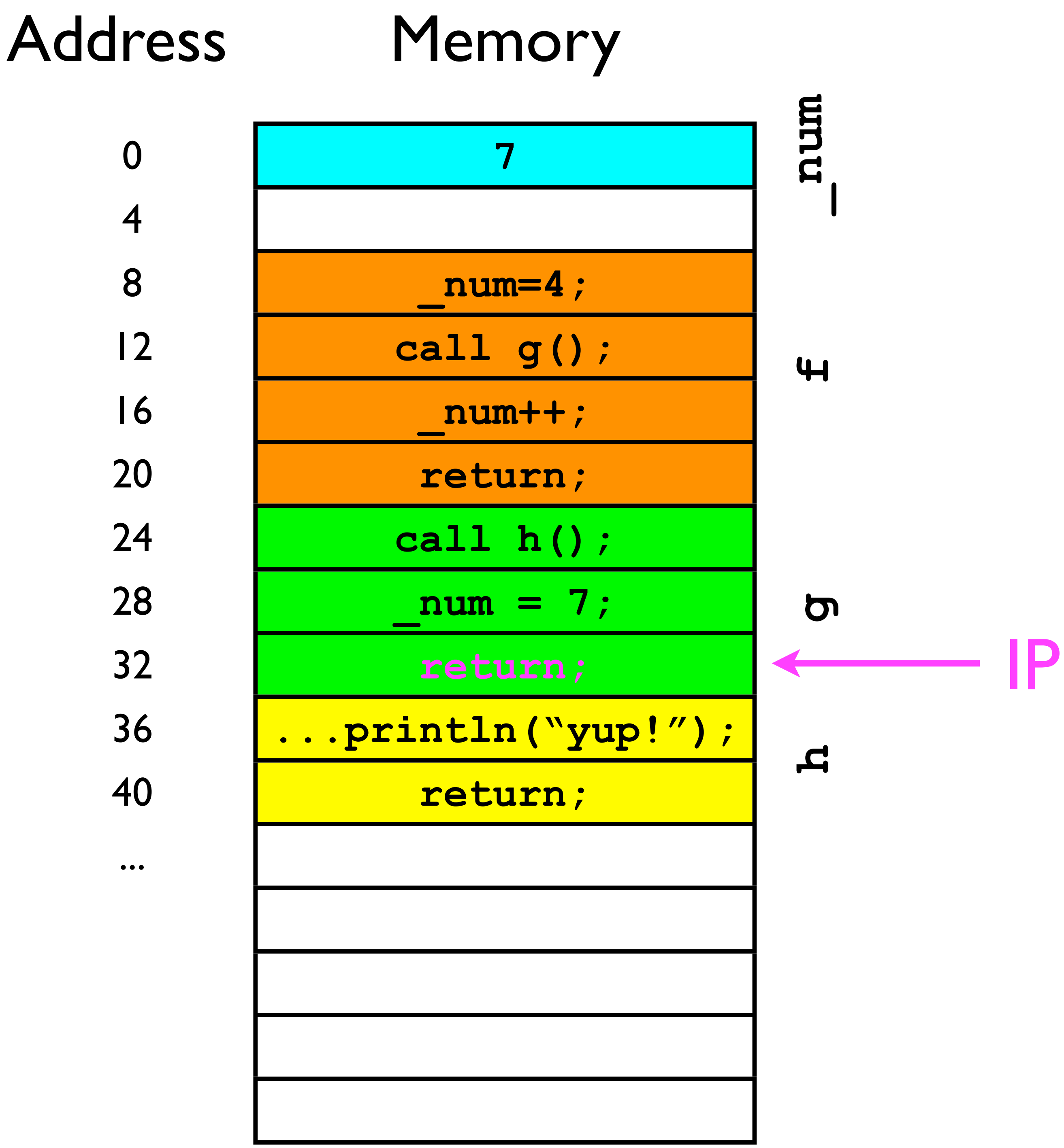
- "Return address" stack:

...and jump to that address.

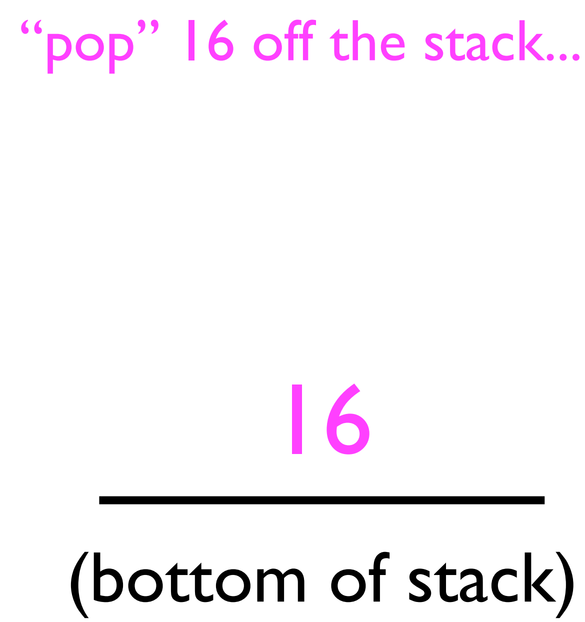
16

(bottom of stack)

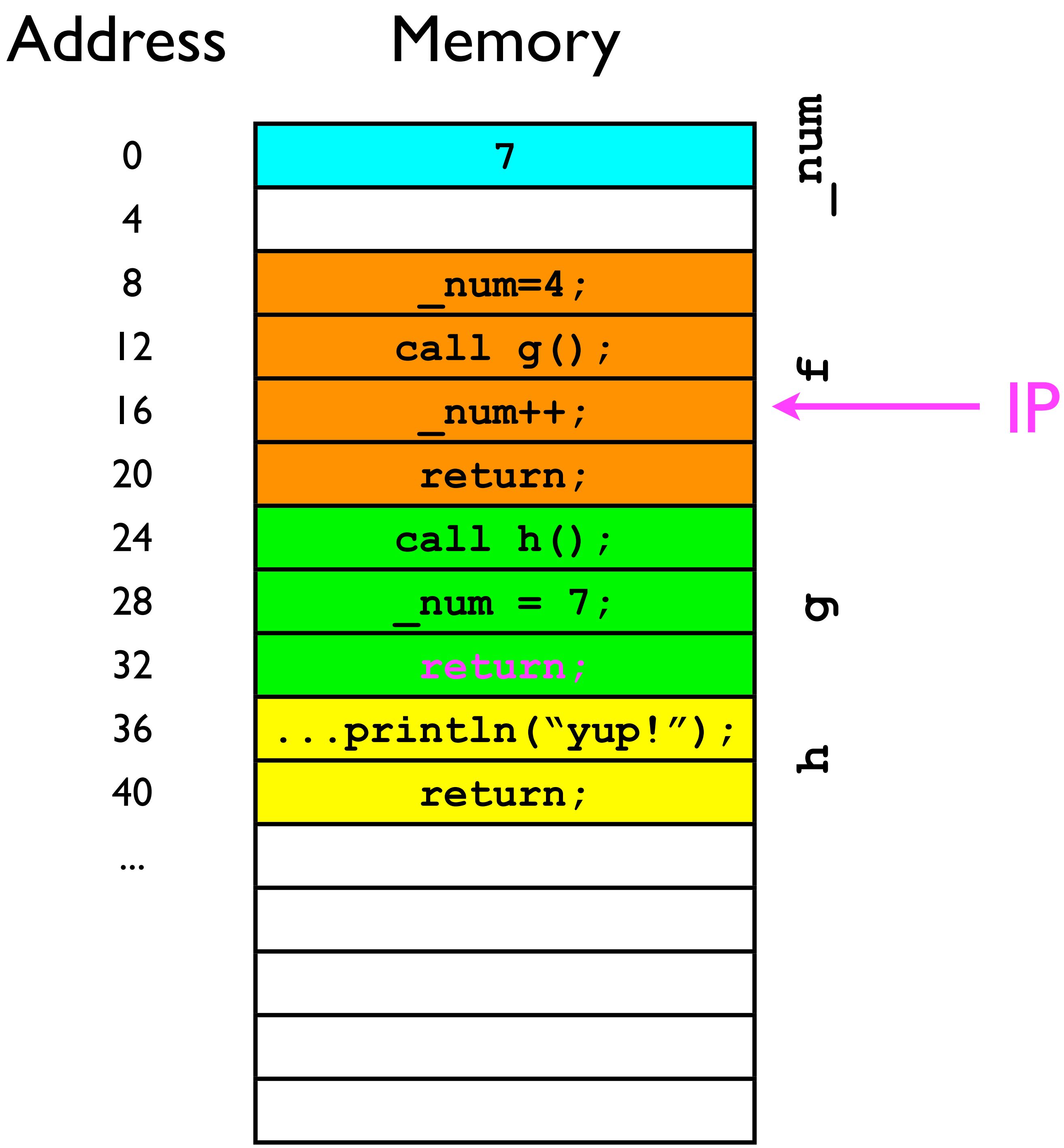
Code execution



- "Return address" stack:



Code execution



- "Return address" stack:

...and jump to that address.

(bottom of stack)

Stack ADT

- To support the last-in-first-out adding/removal of elements, a stack must adhere to the following interface:

```
interface Stack<T> {
  // Adds the specified object to the top of the stack.
  void push (T o);

  // Removes the top of the stack and returns it.
  T pop ();

  // Returns the top of the stack without removing it.
  T peek ();
}
```